ALGORITHM DESIGN ON MULTICORE PROCESSORS FOR MASSIVE-DATA ANALYSIS

A Thesis
Presented to
The Academic Faculty

by

Virat Agarwal

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2010

ALGORITHM DESIGN ON MULTICORE PROCESSORS FOR MASSIVE-DATA ANALYSIS

Approved by:

David A. Bader, Advisor
College of Computing
Georgia Institute of Technology

George Biros
College of Computing
Georgia Institute of Technology

Michael P. Perrone
Multicore Computing Department
IBM T.J. Watson Research Center

Richard Fujimoto
College of Computing
Georgia Institute of Technology

Richard Vuduc
College of Computing
Georgia Institute of Technology

Date Approved: June 23, 2010

To my family.

ACKNOWLEDGEMENTS

I would like to thank my advisor David Bader for giving me excellent guidance and support throughout my career. David first gave me the opportunity for research in High Performance Computing when I spent a summer at the University of New Mexico as a undergraduate research intern after my sophomore year at IIT Delhi. He has always been extremely encouraging, and has guided me to work on challenging and high-impact research problems. I have extremely high regards for him, both professionally and personally. His passion for research and teaching, and his hard-working persona will continue to inspire me.

I am grateful to Michael Perrone, Fabrizio Petrini of IBM TJ Watson Research Center and Davide Pasetto of IBM Ireland, for shaping my overall research direction and giving me the support to complete my dissertation while working with IBM. Fabrizio is a great mentor and an excellent researcher, and I have really enjoyed my time working with him. I have learned a lot of valuable lessons both professionally and personally during my time at IBM, and I thank Michael and Fabrizio for trusting in me and giving me the freedom throughout the last two years.

I am grateful to my proposal and defense committee members – David, George Biros, Rich Vuduc, Richard Fujimoto and Michael Perrone – for monitoring my progress, encouraging my research and reviewing my work.

I thank Carolyn Young, Lometa Mitchell, Arlene Washington, Sheila Williams, Michael Terrell and Alan Glass for taking care of all my school-related administrative stuff so seamlessly!

During my time at Georgia Tech and IBM, I have had the opportunity to collaborate with many amazing researchers, all of whom have taught me a lot. I am especially thankful to Vipin Sachdeva, Kamesh Madduri, Aparna Chandramowlishwaran, Seunghwa Kang, Lurng-Kuo, Hari Subramoni, Daniele Scarpazza, Gordon Braudaway, Karen Magerlein, Ligang, Rebecca Collins, Uri Cummings, Dan Daly, Hubertus Franke, Craig Stunkel, Dale Pearson, Guojing Cong, and others for working with me in the past few years.

My roomates over the years – Amit, Pranay, Varun and Anuj deserve a special thank you, and

I will always remember the time I have spent with them. I am also grateful to the members of the High Performance Computing Lab, and my team at IBM. I thank all my undergrad friends, Neha, Varun, Divya, Rahul, Georgia Tech folks, IBM colleagues, for their support and friendship.

This dissertation is dedicated to my family, Mom, Dad, Tisha, Monali, Sandeep and Nimesh, who have always been there for me, providing their unconditional love and support. Lastly, I must thank Neha for her companionship; and giving me the time full of pleasant memories, her love and support throughout in all my highs and lows. I love you all, and I am fortunate to have you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Analyzing massive-data sets and streams is computationally very challenging. Data sets in systems biology, network analysis and security use network abstraction to construct large-scale graphs. Graph algorithms such as traversal and search are memory-intensive and typically require very little computation, with access patterns that are irregular and fine-grained. The increasing streaming data rates in various domains such as security, mining, and finance leaves algorithm designers with only a handful of clock cycles (with current general purpose computing technology) to process every incoming byte of data in-core at real-time. This along with increasing complexity of mining patterns and other analytics puts further pressure on already high computational requirement. Processing streaming data in finance comes with an additional constraint to process at low latency, that restricts the algorithm to use common techniques such as batching to obtain high throughput.

The primary contributions of this dissertation are the design of novel parallel data analysis algorithms for graph traversal on large-scale graphs, pattern recognition and keyword scanning on massive streaming data, financial market data feed processing and analytics, and data transformation, that capture the machine-independent aspects, to guarantee portability with performance to future processors, with high performance implementations on multicore processors that embed processor-specific optimizations. Our breadth first search graph traversal algorithm demonstrates a capability to process massive graphs with billions of vertices and edges on commodity multicore processors at rates that are competitive with supercomputing results in the recent literature. We also present high performance scalable keyword scanning on streaming data using novel automata compression algorithm, a model of computation based on small software content addressable memories (CAMs) and a unique data layout that forces data re-use and minimizes memory traffic. Using a high-level algorithmic approach to process financial feeds we present a solution that decodes and normalizes option market data at rates an order of magnitude more than the current needs of the market, yet portable and flexible to other feeds in this domain. In this dissertation we discuss in detail algorithm design challenges to process massive-data and present solutions and techniques that we believe can be used and extended to solve future research problems in this domain.

CHAPTER I

INTRODUCTION

Advances in technology are forcing exponential data increase in various domains including financial, scientific, security, and several other application areas. Data that were measured in gigabytes until recently, are now being measured in terabytes, and will soon approach the petabyte range. Streaming data rates over the network are also rapidly increasing to terabits per second and more. To process this magnitude of data, complex techniques, algorithms and models are employed in these domains to analyze, explore, understand and extract meaningful information. The increasing data rates, along with increasing complexity of analytics demand greater computational power which typically comes at a high cost. To fully utilize the computing potential, these applications demand novel and efficient parallel algorithms and high performance implementations.

One intuitive method for analyzing large data sets is through graph abstraction. Graphs constructed using real-world complex networks such as social systems [70, 79, 45, 86], Internet, transportation networks, and biological interaction data [99, 144] can help study analytically several interesting topological and structural aspects of the data. Graphs constructed from these data sets extract unique entities from the data whose interaction is represented by the edges. Depending on the nature of interaction sophisticated edge formulation structures can be designed using interaction attributes. Common graph analysis techniques including degree distribution and diameter expose some structural characteristics of the data set. Designing efficient analytical algorithms to understand both structural and functional characteristics of data is an active research topic [91].

On the other hand, many applications domains in finance and security, requires analyzing and processing streaming data with massive flow rates. Our work, business and personal lives are increasingly relying on critical data that are over the network. Any publicly accessible system on the Internet is a potential target to break-in attempts. These attacks can range from simple e-mail viruses, to general destruction of data, to corporate espionage, to attacks that hijack servers which are used to spread additional attacks in the future. Network Intrusion Detection Systems (NIDS)

identify attacks that use valid packet headers, by searching both headers and payloads to identify attack signatures, and are one of the most promising ways to protect systems on the network. Data analysis in NIDS requires scanning complex knowledge structures on streaming data at real-time. In the financial world also, fueled by the growth of algorithmic and electronic trading, the global options and equities markets are expected to produce an average of more than 128 billion messages a day by 2010, rising from an average of more than 7 billion messages a day in 2007, according to estimates from the TABB Group [138]. The financial institutions need to process this rising magnitude of data streams and perform complex analytics not only at real-time but faster than the other institutions to maintain their competitive advantage. Algorithms in data stream models have been shown effective for statistical analysis and mining trends [101]. Analysis of streaming data is a developing research area that requires a combination of algorithmic innovations with abundance of computing resources to meet the ever increasing demand from the scientific community.

1.1 Research challenges in massive data analysis

Some of the emerging data analytics problems in areas such as systems biology (e.g.disease modeling, modeling reaction and gene regulatory networks), network analysis (e.g. detecting online communities and marketing strategies, studying social interaction networks in epidemiology, egocentric and other networks), and security (e.g. identifying and constructing communities and detecting anomalous trends using mass surveillance networks) utilize network abstraction to construct large scale graphs and perform fundamental analysis queries on them. Graph analysis for these problems is challenging in many respects.

- Current workstations are incapable on handling in-core computations on large graphs due to limited physical memory.

- Graph kernels such as that involve search and traversal typically require very little computation. Most of the work is in fetching data from memory due to random nature of access patterns. The high memory access latencies on current multicore architectures necessitate designing efficient compact data structures and/or access patterns to exploit low latency access from faster memories closer to the processing core.

- Parallelism in these kernels is fine-grained in nature and requires efficient communication and synchronization among processing nodes.

- Graphs arising from real-world networks have unbalanced degree distributions that pose additional design challenges in achieving balanced work partitioning.

- The memory access pattern depends on the graph family and prefetching and caching data on current architectures does not help achieve significant performance improvements.

Data streaming problems in security and mining (e.g. keyword scanning and pattern recognition from network traffic, phone conversations, ATM transactions, web searches, and sensor data) and finance (e.g. automated trading using market data feeds, news feed processing) introduce real-time processing and other challenges to algorithm design.

- At data rates of more than 8 Gigabits per second current architectures at running at 3 GHz frequency have only a handful of clock cycles (less than five with current general purpose computing technology) to perform in-core processing of every incoming byte.

- With the advancement of technology and socio-economic development these streaming data rates are growing rapidly, requiring efficient and scalable parallel solutions to meet processing demands. For instance, the option pricing market data rates, on a one-minute basis, have dramatically increased over the course of the past 4 years, approaching 1 million messages per second. The traffic projection for these feeds is expected to reach an average of more than 14 billion messages a day in 2010.

- The increasing number of complex mining patterns, knowledge structures and analytics puts further pressure on the already high computational requirement. For instance, the latest release of the ClamAV antivirus database has almost half a million signatures that need to be matched with incoming traffic to detect rising complex threats and intrusions.

- Many problems in analyzing streaming data (e.g. keyword/pattern recognition, data mining) construct large state automatas depending on the complexity of the analytics required. The large state automatas are represented by graphs, whose traversal introduces further algorithm design problems as described earlier in this chapter.

- Data stream processing in finance comes with an additional low latency constraint. How fast a trading system can respond to the market will determine who wins and who loses as the first one to identify a trading opportunity generally doesn't leave much behind for the other players in the line. A few milliseconds difference in latency is enough to make the difference. Thus, common techniques of batching used to obtain high throughput cannot be exploited to design solutions in this space.

- Data dependence in streaming data can result in limited available parallelism. With increasing data rates, compression techniques are often employed in data transferred over the network (e.g. financial market data feeds, network data) that introduces a level of inherent sequentialism during processing of incoming feeds.

1.2 Overview of Dissertation and Contributions

With growing interest in data intensive computing, there is a compelling need for innovative algorithms and efficient parallel implementations for high performance data analysis. In this dissertation, we design and analyze efficient parallel algorithms, that can process massive data sets and streams, from several application areas in network analysis, security and computational finance. To develop high performance solutions for these applications we tap in to the potential of the emerging multicore architectures, that are likely to become the building blocks of future exascale systems, and optimize our algorithms to obtain parallel speedup. The dissertation is organized as follows:

In Chapter 2, we present new parallel algorithms for the Breadth-First search (BFS) [6] and List ranking [20, 21] problems. These are representative kernels of many memory intensive combinatorial applications and valuable benchmarks for evaluating the performance of novel architectures for graph theoretic problems. We investigate the challenges involved in exploring very large graphs by designing a breadth-first search (BFS) algorithm for advanced multi-core processors. Our new methodology for large-scale graph analytics combines a high-level algorithmic design that captures the machine-independent aspects, to guarantee portability with performance to future processors, with an implementation that embeds processor-specific optimizations. We present an extensive experimental study that uses state-of-the-art Intel multicore processors. Our performance on several benchmark input data sets representative of the power-law graphs found in real-world problems

reaches processing rates that are competitive with supercomputing results in the recent literature. We also present in this chapter an efficient parallel algorithm for List ranking on the Sony-Toshiba-IBM Cell Broadband Engine (Cell B.E.), a heterogenous multicore architecture that is the core of the Sony Playstation 3, using a generic work partitioning technique. This is the first result that proves the the Cell B.E. works well for workloads that exhibit irregular memory access patterns. Our techniques can be extended to parallelize other graph-theoretic algorithms on this processor.

Chapter 3 is focused on designing algorithms for analyzing streaming data for security and mining. String and pattern matching is one of the most compute intensive steps in a network intrusion detection system and also widely applicable to other scientific domains such as image analysis, speech recognition as well as computer-aided diagnosis systems. The growing network rates, rapidly approaching terabits per second, and the large number of signatures that need to be scanned concurrently pose very demanding challenges to algorithmic design and practical implementation. A practical solution to the keyword scanning problem not only must be as fast as possible, but also must address many other dimensions:it should be able to parse strings of arbitrary length, should be storage-efficient, should not degrade performance when we increase the number of patterns, should be resilient to attacks, should not waste computational resources, should be portable and exhibit parallel scalability, should allow fast dynamic updates and, above all, should map directly to the native mechanisms of the available hardware to allow for an efficient implementation. We provide a pattern compiler that takes a dictionary and optional training input to generate an automaton, and other optimized data structures, and an efficient run-time system that takes this automaton to parse input data streams [114, 110, 111]. We designed a novel automata compression algorithm, a model of computation based on small software content addressable memories (CAMs) and a unique data layout that forces data re-use and minimizes memory traffic. Our algorithm combines all this into a ground breaking, robust and high performance solution that is both space efficient and resilient to attacks.

Continuing with our research on streaming data analysis, Chapter 4 presents efficient algorithms to process financial market data feeds [4, 3] and analytics [5] to aid high performance trading. With exploding data rates in financial markets, trading systems/ticker plants worldwide also require technological breakthroughs to handle massive data volumes, by parsing, decoding, normalizing and

analyzing them at real-time. We present a novel solution to the decoding and normalization of option market data feeds (from Option Pricing Reporting Authority (OPRA)), that are encoded using the Fix Adapted for STreaming (FAST) protocol, on commodity multicore processors. Our approach captures the essence of OPRA protocol specification in a handful of lines of a high-level descriptive language, thus promising a solution that is high-performance, yet flexible and adaptive. We also present an extensive performance evaluation that exposes important properties of our OPRA protocol parser, and analyzes the scalability of five reference multicore systems, and also provide insight onto the behavior of each architecture trying to explain *where* the time is spent by analyzing, for all the processor architectures under evaluation, each action associated to parser events. This helps determine optimality of our approach, and to evaluate the impact of architectural or algorithmic changes for this type of workload. In Chapter 4 we also design parallel algorithms for financial analytics to accelerate pricing engines. We design, analyze and optimize different high performance pseudo (such as Mersenne Twister) and quasi (such as Hammersley sequence) random number generators and use these kernels design efficient parallel algorithms for European Option pricing.

Data analysis is sometimes more intuitive when transformed into another format. Chapter 5 presents a high performance algorithm to perform Fast Fourier transform (FFT) on the Cell B.E.. Our parallel FFT algorithm [18, 19] uses an iterative out-of-place approach to solve problems with 1K to 16K complex input samples. We describe our methodology to partition the work among the cores of this architecture to efficiently parallelize a single FFT computation. The algorithm is able to utilize the vector hardware units on current multicore processors, and also uses other Cell processor based optimization techniques such as loop unrolling and double buffering. Using an efficient synchronization barrier we present a scalable parallel implementation that obtains a performance improvement of over 4 as we vary the number of processing cores from 1 to 8. Our implementation outperforms several popular implementations of the FFT for this range of complex input samples.

The rest of this chapter describes the architectural details of the systems we have used to optimize our implementations and conduct performance analysis.

1.3 High Performance Computing Systems

Over the last few decades, rapid growth in computational power has enabled exponential performance improvements in software. Moore's law describes this long term trend in the history of computing hardware, in which the number of transistors that can be placed on a integrated circuit has doubled approximately every two years. However due to physical limitations in transistor density and power constraints, we can no longer follow the Moore's law curve. This has resulted in rise of an era of multicore computing platforms, that contain a number of processing cores integrated on a single chip [71, 8, 24, 84, 76, 75]. These architectures characterized by levels of private and shared caches are classically known to be used only for commodity and desktop computing. However, with the availability of increasing computing power, communication resources as well as hardware-integrated data and task parallel units in these architectures they are gaining widespread popularity in the high performance computing industry that generally believes that special purpose solutions form the backbone of parallel computing. Multicore processors from Intel [71] and AMD [8] are ubiquitous in personal computing as well as high-end servers and workstations. The Sony-Toshiba-IBM Cell Broadband Engine (Cell B.E.) [76] is a heterogenous multicore chip built for gaming and media applications. The Sun UltraSparc T1 [84] and T2 [75] processors are multicore designs targeted towards multithreaded workloads and enterprise applications. Larrabee[124] is an upcoming architecture from Intel that contains 80 cores on a single chip.

With the widespread availability of compilers and programming tools for these architectures, porting applications is not that difficult, however, optimizing algorithms to extract maximum performance is very challenging and non-intuitive. This requires application programmers to carefully re-design the high-level structure of the algorithm, identify thread level parallelism, identify smaller building blocks, optimize for cache utilization and memory access, manage communication among sockets and map to the instruction set architecture with detailed assembly level optimizations. Recently publications present efficient implementations that exploit the features of these processors for particle simulation [37], concurrent collections [38], stencil computations [77], gyrokenetic particle-to-grid interpolation [92].

In this dissertation we present results primarily on Cell B.E. and Intel Xeon processors that are

described in more detail in the following sections.

1.3.1 Intel Xeon

In our design and experimental evaluation we have used two Intel Xeon systems: a dual-socket Xeon 5500 (Nehalem-EP) and a four socket Xeon 7560 (Nehalem-EX). Figure 1 provides a visual overview of the system architecture of the two processors, and describes how larger systems with 4 and 8 Nehalem-EX sockets can be assembled to build a shared-memory SMP; table 1 summarizes system parameters.

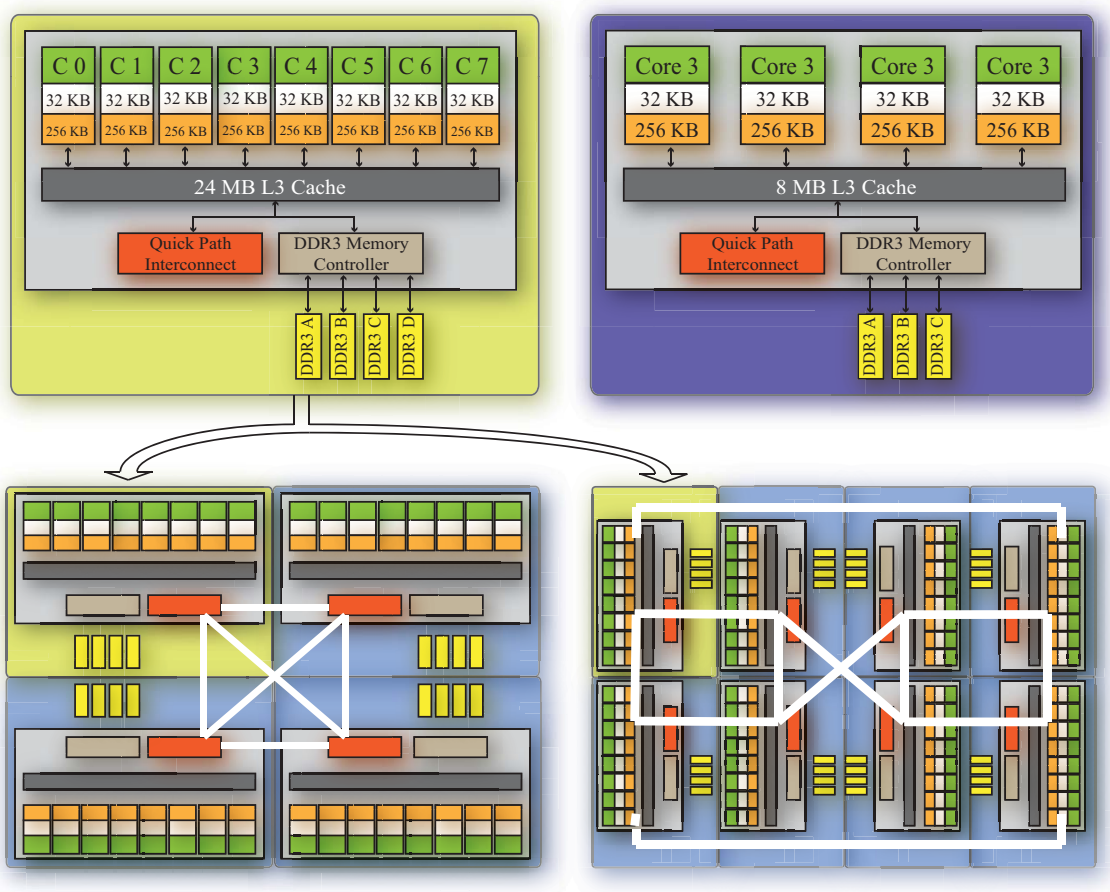

Figure 1: Nehalem EP and EX: architectural overview and topology of 4- and 8-core Nehalem-EX systems.

The Xeon 5570 series (Nehalem-EP) is a 45nm quad core processors. Each core has a private L1 and L2 cache, while the L3 cache is shared across the cores on a socket. Each core supports

Simultaneous Multi Threading (SMT), allowing two threads to share processing resources in parallel on a single core. Nehalem EP has a 32KB L1, 256KB L2 and a 8MB L3 cache. As opposed to older processor configurations, this architecture implements an inclusive last level cache. Each cache line contains 'core valid bits' that specify the state of the cache line across the processor. A set bit corresponding to a core indicates that the core may contain a copy of this line. When a core requests for a cache line that is contained in the L3 cache, the bits specify which cores to snoop, for the latest copy of this line, thus reducing the snoop traffic. Nehalem cache coherency, protocol is MESIF, that extends the MESI protocol to include 'forwarding'. This feature enables forwarding of unmodified data that is shared by two cores to a third one.

The Xeon 7560 (Nehalem-EX), on the other hand, contains 8 cores per socket. The configuration of each core is identical to the Xeon 5600 but they operate at lower frequency. This architecture expands the shared L3 cache size to 24 MB per processor with a fast kilobit-wide ringbus between the different cache segments to boost access speed. Nehalem EX also implements two-way SMT per processing core and contains 4 DDR3 channels per chip, each capable of simultaneous read and write transactions, effectively doubling memory bandwidth. The 4-channel QPI interconnect at 6.4 GigaTransactions per second gives over a 100 GB/sec bandwidth to the other neighbouring sockets in a blade. In Table 1 we can see some interesting architectural trade-offs. The Nehalem EP can run at higher frequencies (high-end EP processors can reach 3.4 GHz), while the top frequency for an EX is only 2.26 GHz. The EP has three DDR3 memory channels for four cores, while the EX has four channels for eight cores, with a less favorable communication to computation ratio.

Table 1: Intel Nehalem system configuration

Processors	Nehalem-EP	Nehalem-EX
Core affinities	Proc 0 : 0-3 & 8-11 Proc 1 : 4-7 & 12-15	Proc 0 : 0-7 & 32-39 Proc 1 : 8-15 & 40-47 Proc 2 : 16-23 & 48-55 Proc 3 : 24-31 & 56-63
Cores per socket	4	8
Core frequency	2.93 GHz	2.26 GHz
L1 cache size	32 KB/32KB	32 KB/32KB
L2 cache size	256KB	256 KB
L3 cache size	8MB	24MB
Cache line size	64 Bytes	64 Bytes
Memory type	3 channels per socket DDR3-1066	4 channels per socket DDR3-1066

1.3.2 IBM Cell Broadband Engine

We have also designed algorithms that exploit the computational resources available on the IBM Cell Broadband Engine (Cell B.E.). This processor is a heterogeneous multi-core chip that is significantly different from conventional multiprocessor or multi-core architectures. It consists of a traditional microprocessor (the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip. Figure 2 gives an architectural overview of the Cell B.E. processor. We refer the reader to [115, 56, 80, 41] for additional details.

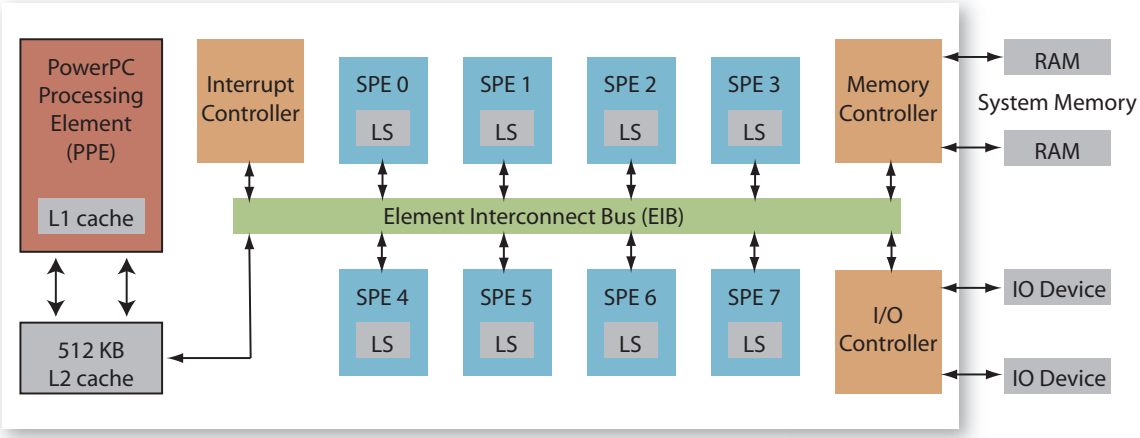

Figure 2: Cell Broadband Engine Architecture.

The PPE runs the operating system and coordinates the SPEs. It is a 64-bit PowerPC core with a vector multimedia extension (VMX) unit, 32 KByte L1 instruction and data caches, and a 512 KByte L2 cache. The PPE is a dual issue, in-order execution design, with two way simultaneous multithreading. Ideally, all the computation should be partitioned among the SPEs, and the PPE only handles the control flow.

Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE. The SPU is a micro-architecture designed for high performance data streaming and data intensive computation. It includes a 256 KByte *local store* (LS) memory to hold SPU program's instructions and data. The SPU cannot

access main memory directly, but it can issue DMA commands to the MFC to bring data into the Local Store or write computation results back to the main memory. DMA is non-blocking so that the SPU can continue program execution while DMA transactions are performed.

The SPU is an in-order dual-issue statically scheduled architecture. Two SIMD [73] instructions can be issued per cycle: one compute instruction and one memory operation. The SPU branch architecture does not include dynamic branch prediction, but instead relies on compiler-generated branch hints using *prepare-to-branch* instructions to redirect instruction prefetch to branch targets. Thus branches should be minimized on the SPE as far as possible.

The MFC supports naturally aligned transfers of 1,2,4, or 8 bytes, or a multiple of 16 bytes to a maximum of 16 KBytes. DMA list commands can request a list of up to 2,048 DMA transfers using a single MFC DMA command. Peak performance is achievable when both the effective address and the local storage address are 128 bytes aligned and the transfer is an even multiple of 128 bytes. In the Cell/B.E., each SPE can have up to 16 outstanding DMAs, for a total of 128 across the chip, allowing unprecedented levels of parallelism in on-chip communication. Kistler *et al.* [80] analyze the communication network of the Cell/B.E. and state that applications that rely heavily on random scatter and or gather accesses to main memory can take advantage of the high communication bandwidth and low latency.

With a clock speed of 3.2 GHz, the Cell processor has a theoretical peak performance of 204.8 GFLOP/s (single precision). The EIB supports a peak bandwidth of 204.8 GB/s for intrachip transfers among the PPE, the SPEs, and the memory and I/O interface controllers. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GB/s to main memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound.

1.3.3 Other systems used for performance comparisons

We have compared our results to the Sun UltraSparc T2 (a.k.a. Niagara-2) processor [75], Cray MTA-2 and Cray XMT [51].

Niagara-2 is a homogeneous multicore server that has 8 cores running at 1.2 GHz each of which is 8 way multithreaded. The cores share a 4MB L2 cache. There are two integer units per core each with a 8 stage pipeline shared among the 8 threads on the core. There is one floating point unit per

core.

The Cray MTA-2 is a multithreaded parallel supercomputer whose architectural features aid in design of parallel graph algorithms. The system we used contained 40 nodes running at 220 MHz each of which is 128 way multithreaded with total of 5120 threads. The system contained a total memory of 160 GB.

Cray XMT is a follow on of the MTA-2 that exhibits the massive multithreaded paradigm. It contains nodes running at 500MHz each of which is 128 way multithreaded. We compared against a system that contains 128 such nodes with a total memory of 4TB on the system.

CHAPTER II

DATA ANALYSIS USING GRAPH TRAVERSAL ON LARGE DATA SETS

Parts of the this chapter appear in :

- V. Agarwal, F. Petrini, D.A. Bader "Scalable graph exploration on Multicore Processors", *Supercomputing (SC '10)*, New Orleans, LA, November, 2010.
- D.A. Bader, V. Agarwal, and K. Madduri, "On the Design and Analysis of Irregular Algorithms on the Cell Processor: A case study on list ranking," *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, CA, March, 2007.
- D.A. Bader, V. Agarwal, K. Madduri, and S. Kang, "High Performance Combinatorial Algorithm Design on the Cell Broadband Engine Processor,", Parallel Computing, 33(10-11):720-740, 2007.

Many areas of science including genomics, astrophysics, artificial intelligence, data mining, national security and information analytics, demand techniques to explore large-scale data sets which are, in most cases, represented by graphs. In these areas, search algorithms are determinant to discover nodes, paths, and groups of nodes with desired properties. Among graph search algorithms, Breadth-First Search (BFS) and List ranking are often used as building blocks for a wide range of graph applications. For example, in the analysis of semantic graphs the relationship between two vertices is expressed by the properties of the shortest path between them, given by a BFS search. Applications in community analysis often need to determine the connected components of a semantic graph [44, 54, 103, 104], and connected components algorithms [105] often employ a BFS search. BFS is the basic building block for best-first search, uniform-cost search, greedy-search and A*, which are commonly used in motion planning for robotics [155, 136]. List ranking [46, 74, 119] is a fundamental combinatorial kernel, and stands in stark contrast to regular scientific computations. It is a key subroutine in parallel graph algorithms for tree contraction and expression evaluation [12], minimum spanning forest [13] evaluation, and ear decomposition. Both

BFS and List ranking also serve as valuable benchmarks and a representative kernels for evaluating the performance of novel architectures. Recognizing the importance of graph theoretic problems in scientific and general-purpose computing, as well as their unique computational and communication characteristics, Asanovic et al. [9] include graph traversal in their list of dwarf kernels.

In this chapter we undertake a comprehensive study of the breadth-first graph traversal problem and describe the challenges we faced to implement an efficient BFS algorithm on a family of Intel Nehalem processors, including a system based on the 8-core Nehalem EX. The choice of a conceptually simple algorithm, such as the BFS exploration, allows for a complete, in-depth analysis of the locality and communication protocols. We also study the problem of list ranking on the IBM Cell B.E. and describe a novel work partitioning that can speed up the performance of other algorithms that exhibit irregular memory access patterns.

The key contributions in this chapter are as follows:

- *A simple and scalable BFS algorithm for multicore, shared-memory systems that can handle efficiently graphs with billions of vertices and beyond.* The main aspects of this algorithm are:
  - an innovative data layout that enhances memory locality and cache utilization through a well-defined hierarchy of working sets;
  - a software design that decouples computation and communication, keeping multiple memory requests in flight at any given time, taking advantage of the hardware capabilities of the Intel Nehalem processors;
  - an efficient, low-latency channel mechanism for inter-socket communication that tolerates the potentially high delays of the cache-coherent protocol;
  - the extreme simplicity: the proposed algorithm relies on a handful of native mechanisms provided by the Linux operating system, (namely `pthreads`, the atomic instructions `__sync_fetch_and_add()` and `__sync_or_and_fetch()`, and the thread and memory affinity libraries).
- *In-depth performance evaluation that considers different classes of real-world graphs* and analyzes scalability, processing rate sensitivity to vertex arity and graph size, and a comprehensive comparative analysis of the related work in graph exploration.

- *Efficient graph traversal on multicore processing using realistic massive graph instances upto billions of edges*: We obtain processing rates that are competitive with supercomputing results in the recent literature. For instance, a 4-socket Nehalem EX is 2.4 times faster than a Cray XMT with 128 processors when exploring a random graph with 64 million vertices and 512 million edges [97].

- *A software path that potentially can be followed by other applications, in particular in the security and business analytics domains.* We expect that many of these hand-crafted techniques described in this chapter will eventually migrate into parallelizing tools and compilers. Processor designers might also find interesting information to develop the new generation of streaming processors, that will likely target from the very beginning the computing needs of streaming and irregular applications. We believe that the results presented in this chapter can be used as algorithmic and architectural building blocks to develop the next generation of exascale machines.

- *An efficient implementation of list ranking on the Cell B.E.*: Using a generic work partitioning technique, we design an algorithm that hides memory latency in irregular access patterns. Our latency-hiding technique, boosts our Cell performance by a factor of about 4.1 for both random and ordered lists. Through the tuning of one parameter in our algorithm, our list ranking code is load-balanced across the SPEs with high probability, even for random lists. The Cell achieves an average speedup of 8 over the performance on current cache-based microprocessors (for input instances that do not fit into the L2 cache). On a random list of 1 million nodes, we obtain a speedup of 8.34 compared to a single-threaded PPE-only implementation. For an ordered list (with stride-1 accesses only), the speedup over a PPE-only implementation is 1.56.

The rest of this chapter describes the essential aspects of our algorithm design followed by a rich body of experimental results.

2.1 BFS Algorithms

In this section we present the methodology that we used to parallelize the BFS algorithm. We first introduce the notation employed throughout the rest of this chapter and a simplified parallel version of BFS. Then, we refine the algorithm and introduce several optimizations that explicitly manage the hierarchy of working sets and enable scaling across cores and sockets.

A graph $G(V,E)$ is composed of a set of vertices V and a set of edges E. Given a graph $G(V,E)$ and a root vertex $r \in V$, the BFS algorithm explores the edges of G to discover all the vertices reachable from r, and it produces a breadth-first tree rooted at r. Vertices are visited in *levels*: when a vertex is visited at level l, it is also said to be at a distance l from the root. When we visit the adjacency list of vertex u, for each vertex v not already visited, we set the parent of v to be u or $P[v] \longleftarrow u$.

The sequential BFS algorithm is a simple linear-time approach that maintains the candidate set of vertices to be explored in a FIFO queue. The queue is initially set to hold the source vertex r. For each vertex v in the queue, its neighbours are inspected and added to the queue if they have not been previously visited. The space and time requirement for sequential BFS are $O(n)$, and $O(m)$, respectively, where n are the number of vertices in the graph and m are the number of edges.

2.1.1 Related Work

The fastest known parallel algorithm for BFS views the graph as an incidence matrix over the semiring, and repeatedly squares this matrix, yielding an algorithm that runs in $O(\log n)$ time using n^3 processors on the concurrent-read concurrent-write (CRCW) model (see [58] for detailed description of the algorithm). In another work [50], this algorithm is extended to yield a $O(\log^2 n)$ time algorithm, that uses $O(n^{2.376})$ processors. The processors requirement make both algorithms impractical for analyzing large-scale graphs.

In recent literature efficient implementations for the BFS have been presented on Cray MTA-2 [22] and Cray XMT [97], that are designed to perform well for graph theoretic workloads. Yoo et al. [154] presented an algorithm based on an efficient edge cut to scale well on the IBM Bluegene/L supercomputer. A recent publication from Scarpazza et al. [123] presents an efficient algorithm for BFS on the IBM Cell processor, that scales well upto a single socket. Successful parallelization

strategies of phylogenetic trees on the Cell processor have also been presented [26]. To avoid memory bandwidth and latency limitations on conventional processors, special purpose hardware [52] has been presented to obtain scalability for applications organized around irregular sparse graphs. Subramaniam et al. [135] present a parallel algorithm that computes BFS tree using reconfigurable meshes. This algorithm uses $O(m)$ processors, where m is the number of edges, thus making it impractical for analyzing large scale graphs. Recently Xia et al. [153] have achieved high-quality results for BFS explorations on state-of-the-art Intel and AMD processors using a topologically adaptive algorithm to determine scalability at each BFS level. Efficient algorithms have also been presented to perform single source shortest paths using a heirachical bucketing structure [139], and randomized approximation algorithms [81]. The randomized algorithm in [81] is based on a high probability transitive closure shortest paths algorithm presented by Ullman and Yannakakis [141] that fails to compute a shortest path only with a low probability. In this chapter we focus on finding an exact breadth first search tree given a root node r.

2.1.2 Simplified Parallel BFS Algorithm

Algorithm 1 presents an initial, simplified parallel BFS algorithm. At any time, CQ (current queue) is the set of vertices that must be visited at the current level. Initially CQ is initialized with the root r (see line 4). At level 1, CQ will contain the neighbours of r, at level 2, it will contain these neighbour's neighbours (the ones that have not been visited in levels 0 and 1), and so on. The algorithm maintains a next queue NQ, containing the vertices that should be visited in the next level. After reaching all nodes in a BFS level, the queues CQ and NQ are swapped. The high-level description Algorithm 1 exposes the nature of the parallelism, but abstracts several important details. For example in the assignment in line 11 must be executed atomically in order to avoid race conditions.

Algorithm 1: Parallel BFS algorithm: high-level overview.

Input: $G(V,E)$, source vertex r

Output: Array $P[1..n]$ with $P[v]$ holding the parent of v

```
1  for all v ∈ V in parallel do
2      P[v] ⟵ ∞;
3  P[r] ⟵ 0;
4  CQ ⟵ Enqueue r;
5  while CQ ≠ φ do
6      NQ ⟵ φ;
7      for all u ∈ CQ in parallel do
8          u ⟵ Dequeue CQ;
9          for each v adjacent to u in parallel do
10             if P[v] = ∞ then then
11                 P[v] ⟵ u;
12                 NQ ⟵ Enqueue v;
13     Swap(CQ, NQ);
```

2.1.3 Designing multicore based parallel BFS algorithm

In order to support algorithmic design, Figure 3 reports the results of a simple benchmark. We consider a collection of data arrays of increasing size, ranging from 4KB to 8GB, that are accessed in read-only mode by a single core using a pseudo-random pattern. The core simply issues a batch of up to 16 memory requests and then waits for the completion of all of them before continuing to the next iteration.

The graphs clearly shows the well known performance impact of the memory hierarchy: a sequence of performance-degrading steps that happen when we overflow one level of cache memory. As expected, by narrowing the working set size to fit in one of caches we can greatly increase performance.

The graphs also shows an important property of the Intel Nehalem processors: we can hide the memory latency by keeping a number of read requests in flight, as traditionally done by multi-threaded architectures [22, 97]. Surprisingly, with a simple software pipelining strategy we can increase by a factor of eight the number of transactions per second: for example, with a working set of 8MB, the memory subsystem can satisfy up to 160 millions reads per second, and with 2 GB we

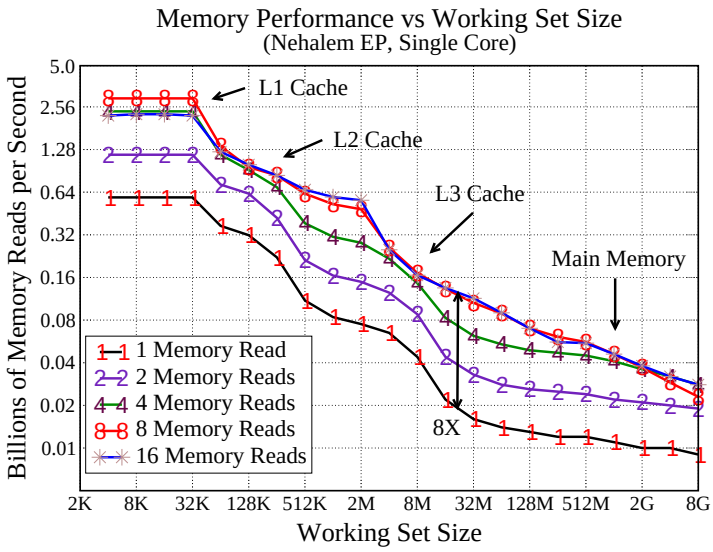

Figure 3: Performance impact of memory pipelining on Nehalem EP

can achieve 40 millions of random reads per second. We run a number of tests and, in accordance with the results presented in [98], we have experimentally determined that the maximum number of outstanding requests is about 10 for both Nehalems EP and EX. When we consider the aggregate behavior of all cores in the socket and we add SMT threads we can keep up to 50 and 75 requests in flight, respectively for the Nehalem EP and EX.[1] So, we can take advantage of one of the pillars of multi-threaded processors with commodity processors.

In Figure 4 we refine the previous experiment by running atomic fetch-and-adds, another important building block of our algorithmic design. In this case we used a buffer of fixed size, 4MB, which is shared by an increasing number of threads mapped on two distinct Nehalem EP sockets.

We observe that atomic ops cannot be pipelined as effectively as memory reads, mostly because the implementation of the primitives relies on the `lockb`[2] assembly instruction that locks the access to part of the memory hierarchy. More interesting is the performance gap when we transition from 4 to 5 threads, crossing the socket boundary. In this case the coherency traffic and the locking instructions limit the scalability of the access pattern: using 8 cores on two sockets, we achieve the same processing rate of only 3 cores on a single socket.

[1] The maximum number of outstanding requests is influenced by the working set size and type of memory operation.

[2] `lockb` is the basic instruction to implement atomic operations on x86 processors, so the results can be directly generalized to other primitives such as read-and-or, that we use in the graph exploration.

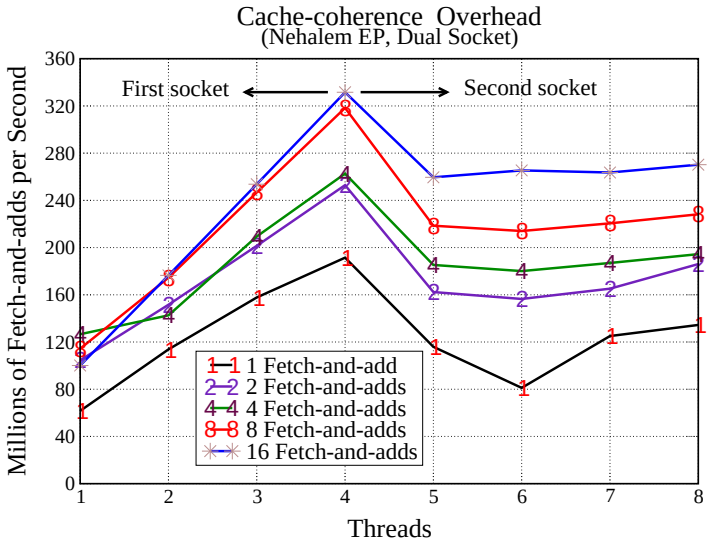

Figure 4: Processing rates with Fetch-and-add and a dual socket configuration.

The insight provided by these two simple experiments is that, while a read-only access pattern can be easily scaled across multiple sockets relying on the native memory pipelining units, more sophisticated patterns that put pressure on the cache-coherency protocol require an innovative algorithmic solution.

2.1.4 Our multicore based parallel BFS Algorithm

Based on the study in previous section, Algorithm 2 refines the original design (given in Algorithm 1) by adding atomic operations (lines 11, 15 and 18) and two important optimizations.

The first optimization is the use of a bitmap (line 4 in Algorithm 2) to mark the vertices during the visit. While the access pattern is still random across all vertices, this greatly reduces the working set size. For example, in 4MB we can store all the visit information for a graph with 32 millions of nodes. In Figure 3, we can see that this can improve the processing rate (number of reads per unit of time) of at least a factor of four.

A more subtle optimization is performed in lines 13 and 14: in this case we avoid the potentially expensive atomic in line 15 *LockedReadSet* (a `__sync_or_and_fetch()` in the real implementation), by first checking whether the vertex has already been visited. It is worth noting that the bit assigned in line 13 may be overwritten by another thread, so to avoid race-conditions we still need to perform an atomic operation. But as shown in Figure 5, the number of atomic operations is

Algorithm 2: Parallel BFS algorithm for a single socket configuration.

Input: $G(V,E)$, source vertex r

Output: Array $P[1..n]$ with $P[v]$ holding the parent of v

```
1  for all v ∈ V in parallel do
2      P[v] ⟵ ∞;
3  for i ⟵ 0..n in parallel do
4      Bitmap[i] ⟵ 0;
5  P[r] ⟵ 0;
6  CQ ⟵ Enqueue r;
7  fork;
8  while CQ ≠ φ do
9      NQ ⟵ φ;
10     while CQ ≠ φ do
11         u ⟵LockedDequeue (CQ);
12         for each v adjacent to u do
13             a ⟵ Bitmap[v];
14             if a = 0 then
15                 prev ⟵ LockedReadSet (Bitmap[v],1);
16                 if prev = 0 then
17                     P[v] ⟵ u;
18                     LockedEnqueue (NQ, v);
19     Synchronize;
20     Swap(CQ, NQ);
21 join;
```

much lower than the number of queries in the later stages of the BFS exploration.

Algorithm 3 includes two important final optimizations. As shown in Figure 4, a random access pattern that requires atomic memory updates cannot scale efficiently across multiple sockets due to heavy traffic for line invalidation and cache locking that will limit the amount of memory pipelining. In order to mitigate this problem we implemented a race-free, lightweight communication mechanism between groups of cores residing on different sockets. Our communication channels rely on two important algorithmic building blocks: an efficient locking mechanism, based on the *Ticket Lock* [132] that protects a lock-free queue based on the *FastForward* algorithm [60]. FastForward, is a cache-optimized single-producer/single-consumer concurrent lock-free queue for pipeline parallelism on multicore architectures, with weak to strongly ordered consistency models. Enqueue

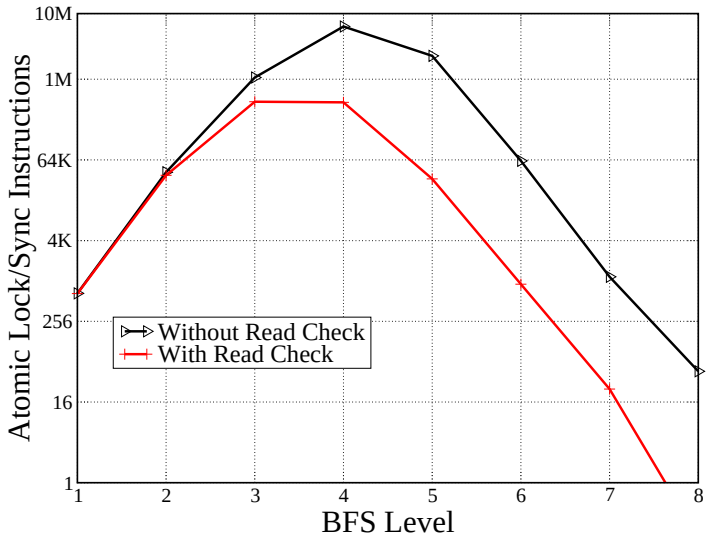

Figure 5: Number of bitmap accesses and atomic operations in a BFS search, uniformly random graph with 16 millions of edges, and average arity 8. By using a simple check we can dramatically reduce the number of atomic operations in the later stages of the exploration.

and dequeue times are as low as 20 nanoseconds on the Nehalem architectures considered in this chapter. One important property of the FastForward queues is that both sender and receiver can make independent progress without generating any unneeded coherence traffic. The activity and the coherency traffic of the FastForward queues can be almost entirely overlapped with the computation when the application is operating in throughput mode, which is the case of our BFS graph exploration.

The final optimization is the use of batching when inserting and removing from the inter-socket communication channel, as shown in Algorithm 3 in lines 29 and 32. Rather than inserting at a granularity of a single vertex, each thread batches a set of vertices to amortize the locking overhead. Overall, including all the synchronization, locking and unlocking and buffer copies, the normalized cost per vertex insertion is only 30 nanoseconds. The inter-socket channels proved to be the key optimization, that allowed us to achieve very good scaling across multiple Nehalem EX sockets.

The overall impact of all optimizations is summarized in Figure 6 for the Nehalem EP. It is worth noting that the change of slope of the most optimized version of the algorithm between 4 and 8 threads is mostly due to the change of algorithm –we rely on a two-phase algorithm, the first one processing the local vertices and the second one processing the remote ones sent through

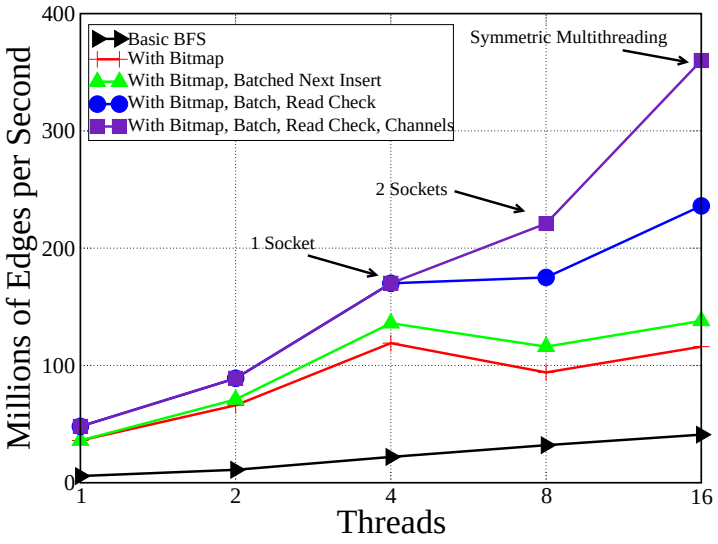

Figure 6: Performance impact of various optimizations on uniformly random graph

the inter-socket communication channels, and not to the communication overhead, given that most operations are overlapped with carefully placed `__mm_prefecth()` instrinsics [82].

The proposed algorithm can be easily generalized to distributed memory machines that use fast and lightweight communication mechanisms, such as PGAS language and libraries [33].

2.1.5 Experimental Results

We tested our algorithm on two different classes of graphs.

- Uniformly Random Graphs : Graphs with n vertices each with degree d, where the d neighbours of a vertex are chosen randomly.

- Scale-free graphs (R-MAT): Graphs generated using the GTgraph [16] suite based on the R-MAT graph model [35] to represent real-world large-scale networks. Using a small number of parameters, R-MAT samples from a Kronecker product to produce scale-free graphs with community structure. These graphs have a few high degree vertices and many low-degree ones.

Table 2 provides the configuration details of the two Intel systems under consideration, a dual-socket Nehalem EP and a four-socket Nehalem EX, and several other parallel systems for BFS discussed in the literature.

Algorithm 3: Multicore Multi Socket Parallel BFS algorithm

Input: $G(V,E)$, source vertex r

Output: Array $P[1..n]$ with $P[v]$ holding the parent of v

```
1  sockets = GetTotalSockets();
2  Partition graph, allocate n/sockets nodes on each socket ;
3  for s ⟵ 0 to sockets do
4      Allocate memory for P[s] on s;
5      Allocate memory for Bitmap[s] on s;
6      for v ⟵ 0 to n/sockets do
7          P[s][v] ⟵ ∞;
8          Bitmap[s][v] ⟵ 0;
9  P[DetermineSocket(r)][r] ⟵ 0;
10 for s ⟵ 0 to sockets do
11     CQ[s] ⟵ φ;
12     NQ[s] ⟵ φ;
13 CQ[DetermineSocket(r)] ⟵ Enqueue r;
14 fork;
15 this = GetMySocket();
16 while CQ[this] ≠ φ do
17     while CQ[this] ≠ φ do
18         u ⟵LockedDequeue (CQ[this]);
19         for each v adjacent to u do
20             s ⟵ DetermineSocket(v);
21             if s = this then
22                 a ⟵ Bitmap[this][v];
23                 if a = 0 then
24                     prev ⟵ LockedReadSet (Bitmap[this][v],1);
25                     if prev = 0 then
26                         P[v] ⟵ u;
27                         LockedEnqueue (NQ[this], v);
28             else
29                 LockedEnqueue (SQ(s),(v, u));
30     Synchronize;
31     while SQ(this) ≠ φ do
32         (v, u) ⟵ LockedDequeue (SQ(this));
33         a ⟵ Bitmap[this][v];
34         if a = 0 then
35             prev ⟵ LockedReadSet (Bitmap[this][v],1);
36             if prev = 0 then
37                 P[v] ⟵ u;
38                 LockedEnqueue (NQ[this], v);
39     Synchronize;
40     Swap(CQ[this], NQ[this]);
41     NQ[this] ⟵ φ;
42 join;
```

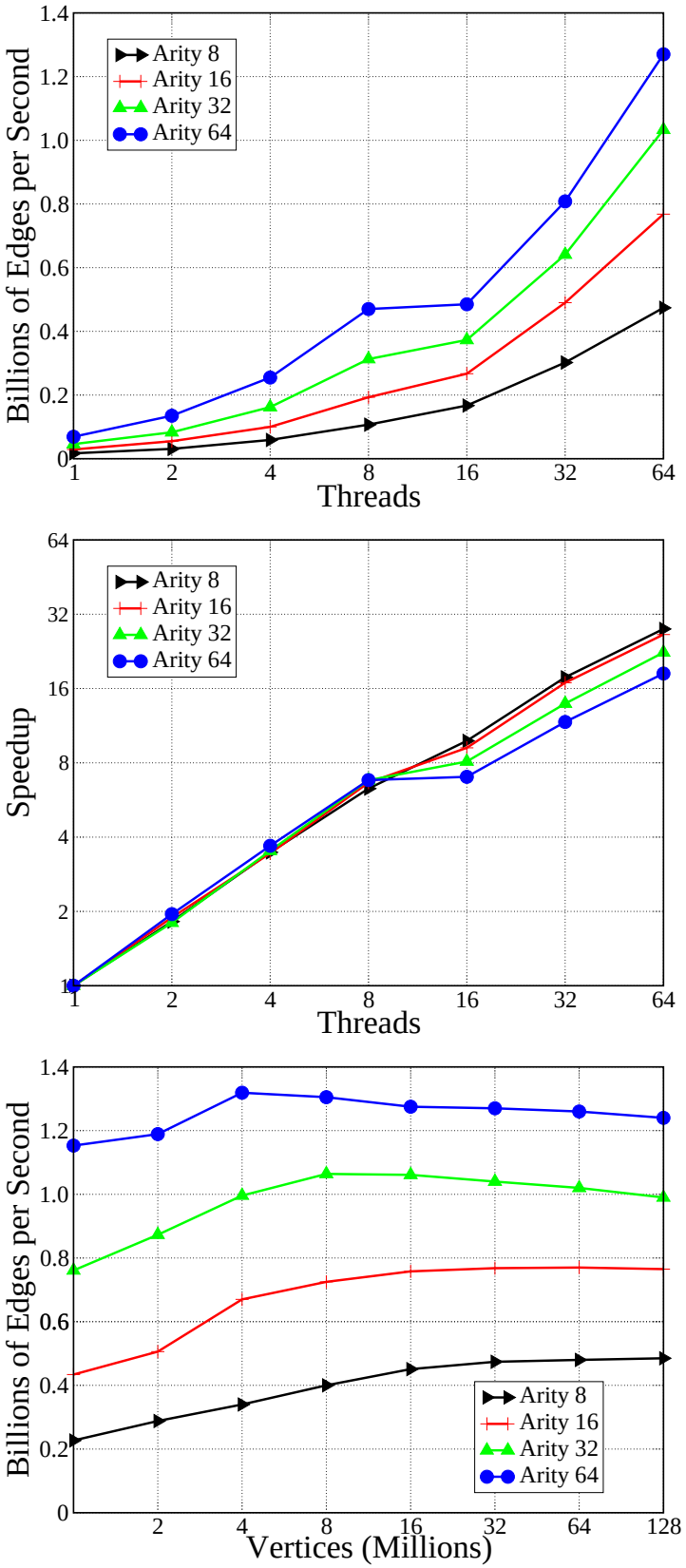

Figure 7: Performance of our parallel BFS algorithm for uniformly random graphs on Nehalem EP

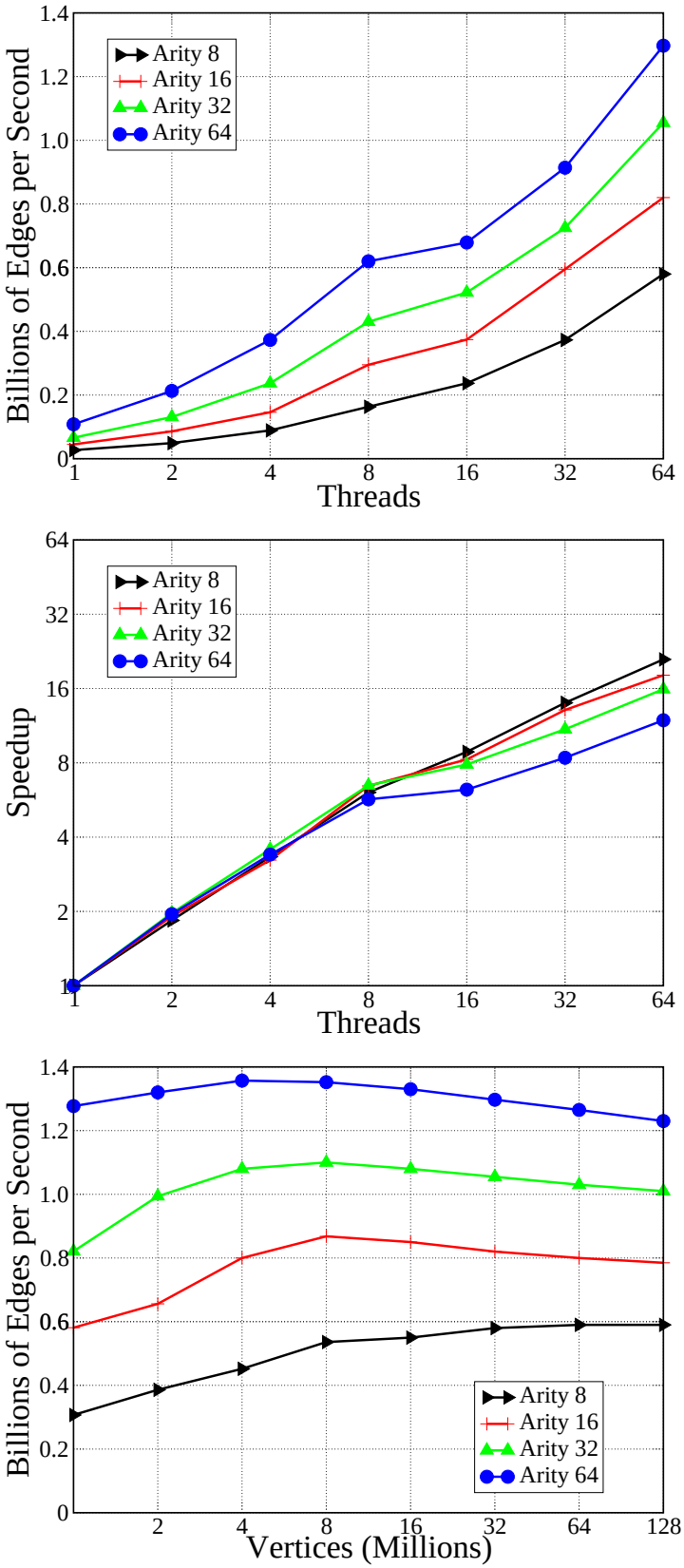

Figure 8: Performance of our parallel BFS algorithm for RMAT graphs on Nehalem EP

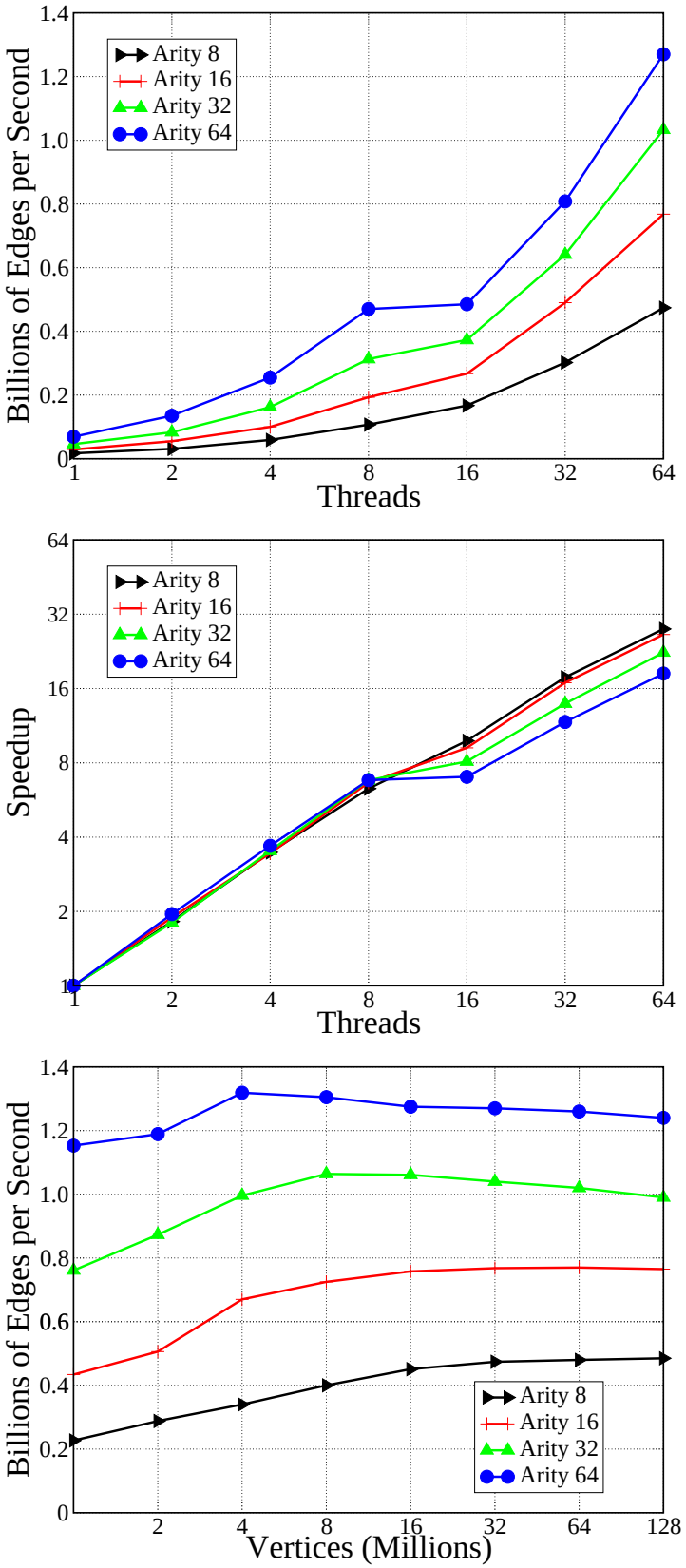

Figure 9: Performance of our BFS algorithm for uniformly random graphs on Nehalem EX

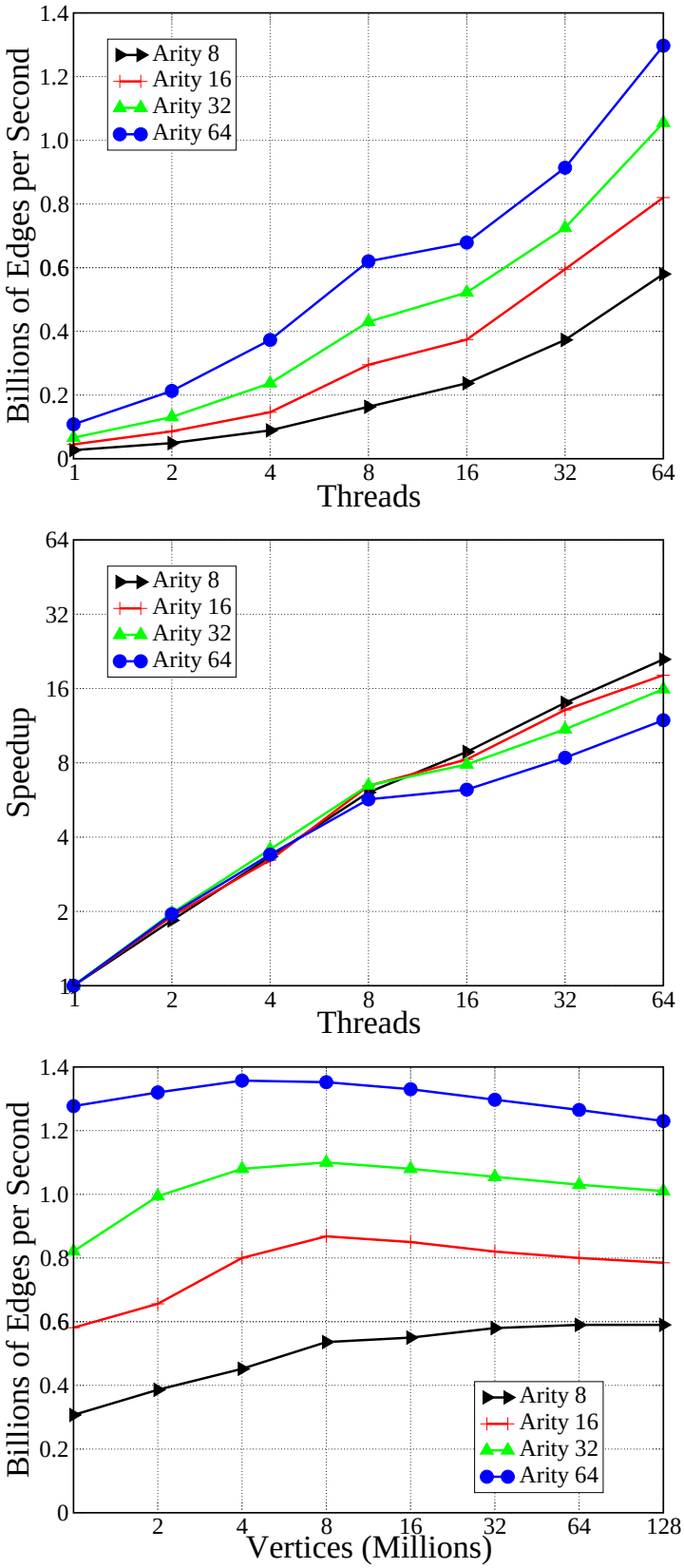

Figure 10: Performance of our BFS algorithm for RMAT graphs on Nehalem EX

Table 2: Configuration details of the various architectures used for performance comparison of our parallel breadth-first search algorithm

	CPU Speed (GHz)	Sockets	Cores /Socket	Threads /Core	Threads	Cache /Socket	Cache	Memory
INTEL Xeon 7500 (Nehalem EX)	2.26	4	8	2	64	24M	96M	256G
INTEL Xeon X5570 (Nehalem EP)	2.93	2	4	2	16	8M	16M	48G
INTEL Xeon X5580 (Nehalem EP)	3.2	2	4	2	16	8M	16M	16G
CRAY XMT	500MHz	128	-	-	16K			1TB
CRAY MTA-2	220MHz	40	-	-	5120			160G
AMD Opteron 2350 (Barcelona)	2.0	2	4	1	8	2M	4M	16G

We measure the performance of the proposed BFS algorithm in edges processed per second. This is computed using $\frac{m_a}{running\ time}$, where m_a is the actual number of edges traversed during the BFS computation ($m_a < m$ for graphs that are not fully connected, where m is the total number of edges in the graph). In our measurements we noticed a maximum difference of 2% between m and m_a. The source vertex was chosen randomly in all the experiments given in this section.

Figures 7(a) & 7(b) plot the processing rate and speedup obtained on a uniformly random graph with 32 million vertices when the number of edges varies from 256 million to 1 billion, and number or threads varies from 1 to 16 on a Nehalem EP. The speedup here is defined as the ratio of processing rate on t threads over 1 thread. We used the best performing algorithm for each thread configuration. When the threads run on the same socket, we disable inter socket channels to get the highest performance and when they run across sockets we use the channels to get best performance. Here we use one thread per core up to 8 threads and use SMT to scale to 16 threads. We see that the performance of our algorithm not only scales well to multiple cores on the same socket but also across the two sockets of an EP system. Figure 7(c) plots the maximum processing rate obtained on a 2-socket Nehalem EP for uniformly random graphs when the number of edges varies from 256 million to 1 billion and the number of vertices varies from 1 million to 32 million. We observe in this plot that the processing rate only drops by a small factor with increasing number of vertices. This is due to the higher random access latency with increasing working set sizes. We obtain a processing rate between 200 to 800 million edges per second on a Nehalem EP with 2 sockets for both uniformly random and R-MAT graphs with 32 million vertices when the number of edges is

varied from 256 million to 1 billion.

Similarly, Figure 8 plots the processing rate and speedup of R-MAT graphs on a Nehalem EP. Here we notice that the R-MAT graphs have higher processing rates than uniformly random graphs. This is because R-MAT graphs have a few high degree vertices that lead to a performance advantage more than the performance degradation caused by the low degree vertices in the graph. The slight reduction in the slope of the speedup curve from 4 to 8 threads is because of the change in the algorithm used when running on 1 socket vs 2 sockets. The multi-socket algorithm performs extra insert/delete operations on inter-socket communication channels. We obtain a constant slope speedup curve, when the same channel-based algorithm is used on a single socket.

Figures 9 & 10 show the processing rate and speedup of uniformly random and R-MAT graphs on a 4 socket Nehalem EX. Here we notice that the performance of our algorithm scales well upto all 64 threads (32 cores) available on this blade, giving a speedup between 14-24 over the performance of a single thread. Again, as we noticed before, here the slope of the speedup curve slightly reduces from 8 to 16 threads, when the algorithm starts using inter-socket channels for task division and communication. We also note from Figures 9(c) & 10(c) show that the processing rate is less affected by the number of vertices in the graph. This is due to a larger cache size on the Nehalem EX. We obtain a processing rate from 0.55 to 1.3 billion edges per second on 4 sockets of this processor for both uniformly random and R-MAT graphs with 32 million vertices when the number of edges varies from 256 million to 1 billion.

We provide a reference table (Table 3) that summarizes several published results on parallel breadth first search. These results are categorized based on the graph size/types used, scalability with processors and with graph size, and choose some selected performance results that can be compared with the performance of our algorithm. The performance column in this table gives the performance in million edges per second (ME/s) for a graph with N vertices and M edges. When compared with these results our BFS algorithm running on 4-socket Nehalem EX is:

- 2.4 times faster than a Cray XMT [97] with 128 processors when exploring a uniformly random graph with 64 million vertices and 512 million edges.
- 5 times faster than 256 BlueGene/L [154] processors on a graph with average degree 50.

Table 3: Related work on parallel breadth-first search

Reference	Graph (size) type	Performance					Base architecture	Scalability (cores)	Scalability (graph size)
		Type	N (vertices)	M (edges)	ME/s	p processors			
Bader, Madduri [22]	(Large) Random, R-MAT, SSCA	R-MAT SSCA2v1 SSCA2v1	200M 32M 4M	1B 310M 512M	500 250 250	40 10 10	Cray MTA-2	Good	Good
Mizell, Maschhoff [97]	(Large) Uniformly Random	Uniformly Random	64M	512M	210	128	Cray XMT	Good	Excellent
Scarpazza, Villa, Petrini [123]	(Medium) Uniformly Random	Uniformly Random Uniformly Random Uniformly Random Uniformly Random	25 5M 2.5M 1M	256M 256M 256M 256M	101 305 420 540	1 chip 1 chip 1 chip 1 chip	IBM Cell/B.E.	Good	Poor
Yoo et al. [154]	Large	- - - -	Peak Peak Peak Peak	$d10$ $d50$ $d100$ $d200$	80 232 492 731	256 256 256 256	IBM BlueGene/L	Good	Good
Xia, Prasanna [153]	(Small) Uniformly Random, Grids	8-Grid 16-Grid	1M 1M	16M 32M	220 311	Peak Peak	dual Intel X5580	Moderate	Moderate

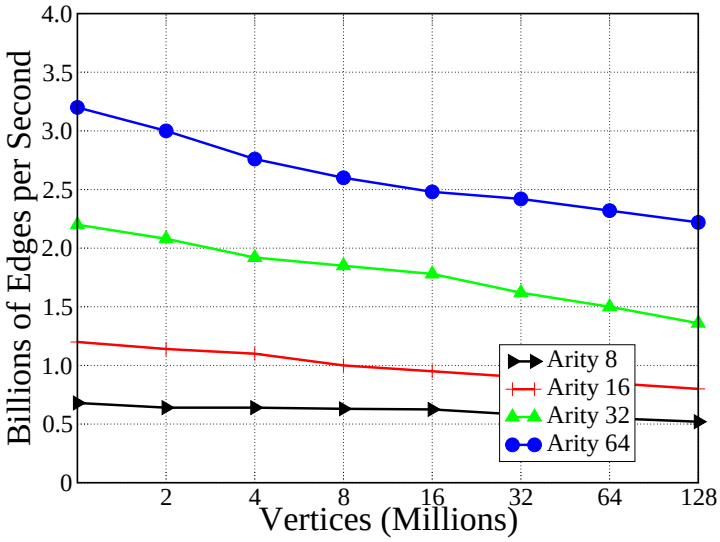

Figure 11: Throughput performance of our parallel BFS algorithm for uniformly random graphs on Nehalem EX

- capable of processing 550 million edges per second with an R-MAT graph with 200 million vertices and 1 billion edges, comparable to the performance of a similar graph on a Cray MTA-2 [22] with 40 processors.

Figure 11 plots the throughput of our algorithm, where we run a single BFS per socket and run multiple instances of the algorithm on different graphs on different sockets. This is representative of the SSCA#2 benchmarks, and gives an estimate on the performance of our algorithm for such workloads.

2.2 List ranking

Given an arbitrary linked list that is stored in a contiguous area of memory, the list ranking problem determines the distance of each node to the head of the list. For a random list, the memory access patterns are highly irregular, and this makes list ranking a challenging problem to solve efficiently on parallel architectures. Implementations that yield parallel speedup on shared memory systems exist [65, 14], yet none are known for distributed memory systems.

It is general perception that the Cell architecture is not suited for problems that involve fine-grained memory accesses, and where there is insufficient computation to hide memory latency. The *list ranking problem* [46, 119, 74] is representative of such problems, and is a fundamental paradigm

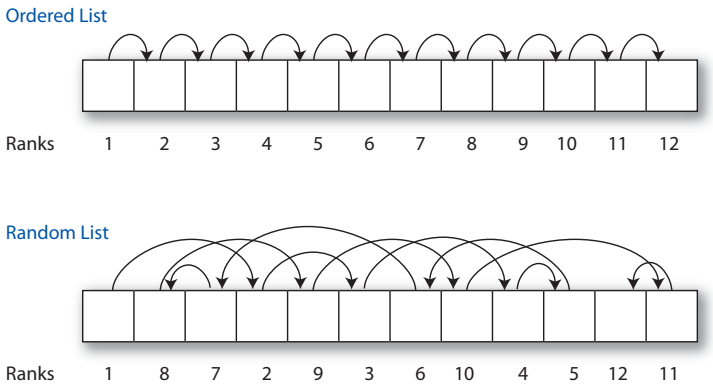

Figure 12: List ranking for ordered and random list

for the design of many parallel combinatorial and graph-theoretic applications. Using list ranking, fast parallel algorithms for shared memory computers have demonstrated speedups compared with the best sequential implementation for graph theoretic problems such as ear decomposition [15], tree contraction and expression evaluation [17], spanning tree [12] and minimum spanning forest [13].

List ranking is an instance of the more general prefix problem [14]. Let X be an array of n elements stored in arbitrary order. For each element i, let $X(i).$*value* denote its value and $X(i).$*next* the index of its successor. Then for any binary associative operator $\oplus$, compute $X(i).$*prefix* such that $X(head).$*prefix* $= X(head).$*value* and $X(i).$*prefix* $= X(i).$*value* $\oplus\, X(predecessor).$*prefix*, where *head* is the first element of the list, i is not equal to *head*, and *predecessor* is the node preceding i in the list. If all values are 1 and the associative operation is addition, then prefix reduces to list ranking. We assume that we know the location of the head h of the list, otherwise we can easily locate it. The parallel algorithm for a canonical parallel computer with p processors is as follows:

1. Partition the input list into s sublists by randomly choosing one node from each memory block of $n/(s-1)$ nodes, where s is $\Omega(p \log n)$. Create the array *Sublists* of size s.

2. Traverse each sublist computing the prefix sum of each node within the sublists. Each node records its sublist index. The input value of a node in the *Sublists* array is the sublist prefix sum of the last node in the previous *Sublists*.

3. The prefix sums of the records in the *Sublists* array are then calculated.

4. Each node adds its current prefix sum value (value of a node within a sublist) and the prefix sum of its corresponding *Sublists* record to get its final prefix sums value. This prefix sum value is the required label of the leaves.

2.2.1 List ranking on Cell

2.2.1.1 A Novel Latency-hiding Technique for Irregular applications

Due to the limited local store (256 KB) within a SPE, memory-intensive applications that have irregular memory access patterns require frequent DMA transfers to fetch the data. The relatively high latency of a DMA transfer creates a bottleneck in achieving performance for these applications. Several combinatorial problems, such as the ones that arise in graph theory, belong to this class of problems. Formulating a general strategy that helps overcome the latency overhead will provide direction to the design and optimization of irregular applications on Cell.

Since the Cell supports non-blocking memory transfers, memory transfer latency will not be a problem if we have sufficient computation between a request and completion. However, if we do not have enough computation in this period (for instance, the Helman-JáJá list ranking algorithm [65]), the SPE will stall for the request to be completed. A generic solution to this problem would be to re-structure the algorithm such that the SPE keeps doing useful computation until the memory request is completed. This essentially requires identification of an additional level of parallelism/concurrency within each SPE. Note that if the computation can be decomposed into several independent tasks, we can overcome latency by exploiting concurrency in the problem.

Our technique is analogous to the concept of tolerating latency in modern architectures using thread-level parallelism. The SPE does not have support for hardware multithreading, and so we manage the computation through *software-managed threads*. The SPE computation is distributed to a set of software-managed threads (SM-Threads) and at any instant, one thread is active. We keep switching software contexts so that we do computation between a DMA request and its completetion. We use a round-robin schedule for the threads.

Through instruction-level profiling, it is possible to determine the minimum number of SM-Threads that are needed to hide the memory latency. Note that utilizing more SM-Threads than required also incurs an overhead. Each SM-Thread introduces additional computation and also

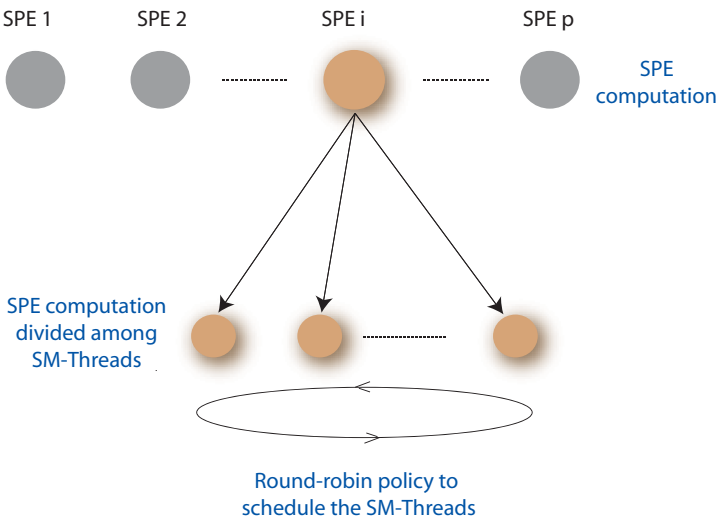

Figure 13: Illustration of the round-robin scheduling technique

requires memory on the limited local store. Thus, we have a trade-off between the number of SM-Threads and latency due to DMA stalls. In the next section, we will use this technique to efficiently implement list ranking on Cell.

2.2.1.2 *Parallel List ranking implementation on Cell*

Our Cell implementation (described in high-level in the following four steps) is similar to the Helman–JáJá algorithm [65]. Let us assume p SPEs in the analysis.

1. We uniformly pick s *head nodes* in the list and assign them to the SPEs. So, each SPE will traverse s/p sublists.

2. Using these s/p sublists as independent SM-Threads, we adopt the latency-hiding technique. We divide the s/p sublists into b DMA list transfers. Using one DMA list transfer, we fetch the next elements for a set of s/pb lists. After issuing a DMA list transfer request, we move onto the next set of sublists and so forth, thus keeping the SPU busy until this DMA transfer is complete. Figure 14 illustrates step 3 of this algorithm.

   We maintain temporary structures in the Local Store (LS) for these sublists, so that the LS can create a contiguous sublist out of these randomly scattered sublists, by creating a chain of next elements for the sublists.

   After one complete round, we manually revive this SM-Thread and wait for the DMA transfer

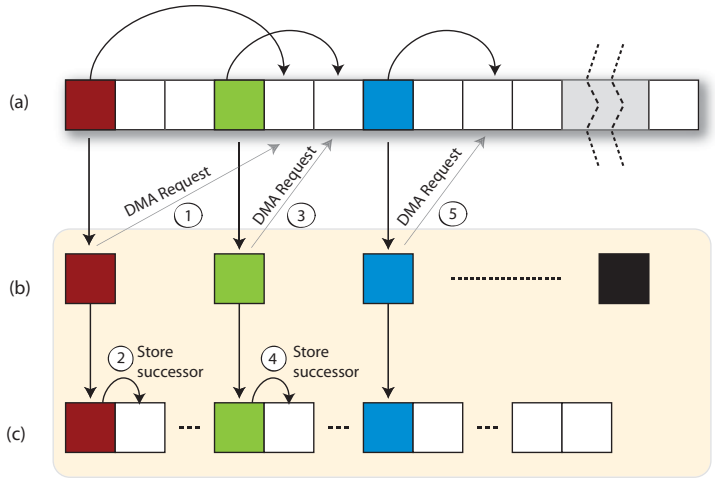

Figure 14: Step 2 of List ranking on Cell. (a) Linked list for which list ranking is to be done. Colored nodes here are allocated to SPE(i), (b) View from SPE(i), it has s/p sublist head nodes to traverse concurrently, (c) This array is used to store sublists in contiguous area of memory. When this gets full, we transfer it back to the main memory.

to complete. Note that there will be no stall if we have sufficient number of SM-Threads (we determine this number in Section 2.2.2) to hide the latency. We store the elements that are fetched into the temporary structures, initiate a new DMA list transfer request for fetching the successors of these newly fetched elements, and move on to the next set of sublists.

When these temporary structures get full, we initiate a new DMA list transfer request to transfer back these elements to the main memory.

At the end of Step 2, we have the prefix sum of each node within the sublist for each sublist within the SPU. Also, we have the randomly scattered sublists stored into a contiguous area of memory.

3. Compute the rank of each sublist head node using the PPU.

The running time for step 2 of the algorithm dominates over the rest of algorithm by an order of magnitude. In the asymptotic notation, this step is O(n). It consists of an outer loop of O(s) and an inner loop of O(*length of the sublist*). Since the lengths of the sublists are different, the amount of work performed by each SM-Thread differs. For a large number of threads, we get sufficient computation for the SPE to hide DMA latency even when the load is imbalanced. Helman and JáJá [65, 66] established that with high probability, no processor would traverse more $\alpha(s)\frac{n}{p}$ elements

for $\alpha(s) \geq 2.62$. Thus, the load is balanced among various SPEs under this constraint. In our implementation, we incorporate recommended software strategies [29] and techniques to exploit the architectural features of Cell. For instance, we use manual loop unrolling, double buffering, branch hints, and design our implementation for a limited local store.

2.2.2 Experimental Results

We report our performance results from runs on a IBM BladeCenter QS20, with two 3.2 GHz Cell BE processors, 512 KB Level 2 cache per processor, and 1 GB memory (512 MB per processor). We use one processor for measuring performance and compile the code using the gcc compiler provided with Cell SDK 1.1, with level 3 optimization.

Similar to [65, 14] we use two classes of lists to test our code, *Ordered* and *Random*. An ordered list representation places each node in the list according to its rank. Thus node i is placed at position i, and its successor is at position $i + 1$. A random list representation places successive elements randomly in the array.

Our significant contribution to this chapter is a generic work partitioning technique to hide memory latency. We demonstrate the results of this technique for list ranking: we use DMA-level parallelism to vary the number of outstanding DMA requests on each SPE, as well as partition the problem and allocate more sublists to each SPE. Figure 15 shows the performance boost we obtain as we tune the DMA parameter. From instruction level profiling of our code we determine that the exact number of computational clock cycles between a DMA transfer request and its completion are 75. Comparing this with the DMA transfer latency (90ns, i.e. about 270 clock cycles) suggests that four outstanding DMA requests should be sufficient for hiding the DMA latency. Our results confirm this analysis and we obtain an improvement factor of 4.1 using 8 DMA buffers.

In Figure 16 we present the results for load balancing among the 8 SPEs, as the number of sublists are varied. For ordered lists, we allocate equal chunks to each SPE. Thus, load is balanced among the SPEs in this case. For random lists, since the length of each sublist varies, the work performed by each SPE varies. We achieve a better load balancing by increasing the number of sublists. Figure 16 illustrates this: load balancing is better for 64 sublists than the case of 8 sublists.

We present a performance comparison of our implementation of list ranking on Cell with other

single processor and parallel architectures. We consider both random and ordered lists with 8 million nodes.

Figure 17 shows the running time of our Cell implementation compared with efficient implementations of list ranking on the following architectures:

Intel_x86: 3.2 GHz Intel Xeon processor, 1 MB L2 cache, Intel C compiler v9.1. **Intel_i686:** 2.8 GHz Intel Xeon processor, 2 MB L2 cache, Intel C compiler v9.1.

Intel_ia64: 900 MHz Intel Itanium 2 processor, 256 KB L2 cache, Intel C compiler v9.1.

SunUS_III: 900 MHz UltraSparc-III processor, Sun C compiler v5.8.

MTA-[1,2,8]: 220 MHz Cray MTA-2 processor, no data cache. We report results for 1,2,8 processors.

SunUS-[1,2,8]: 400 MHz UltraSparc II Symmetric Multi-processor system (Sun E4500), 4 MB L2 cache, Sun C compiler. We report results for 1,2 and 8 processors.

Finally, we demonstrate a substantial speedup of our Cell implementation over a sequential implementation using the PPE only. We compare a simple pointer-chasing approach to our algorithm using different problem instances. Figure 18 shows that for random lists we get an overall speedup of 8.34 (1 million vertices), and even for ordered lists we get a speedup of 1.5.

2.3 Summary

In this chapter we present a scalable breadth-first search (BFS) algorithm for multicore processors. In spite of the highly irregular access pattern of the BFS, our algorithm is able enforce various degrees of memory and processor locality, minimizing the negative effects of the cache-coherency protocol between processor sockets. The experimental results, conducted on two Nehalem platforms, a dual-socket Nehalem EP and a four-socket Nehalem EX, demonstrate an impressive processing rate in parsing graphs that have up to a billion edges. Using several graph configurations the Nehalem EX system reaches, and in many cases exceeded, the performance of special-purpose supercomputers designed to handle irregular applications.

We also present a generic work partitioning technique to hide memory latency on Cell. This

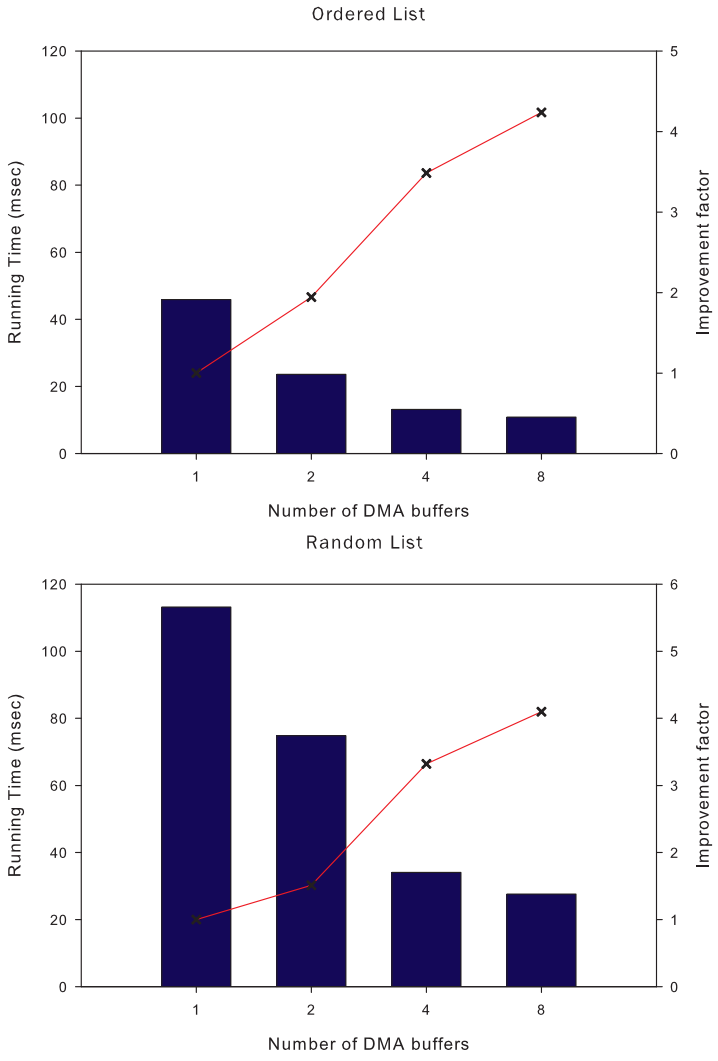

Figure 15: Achieving Latency Tolerance for Step 2 of the List ranking algorithm through DMA parameter tuning, for lists of size 2^{20}

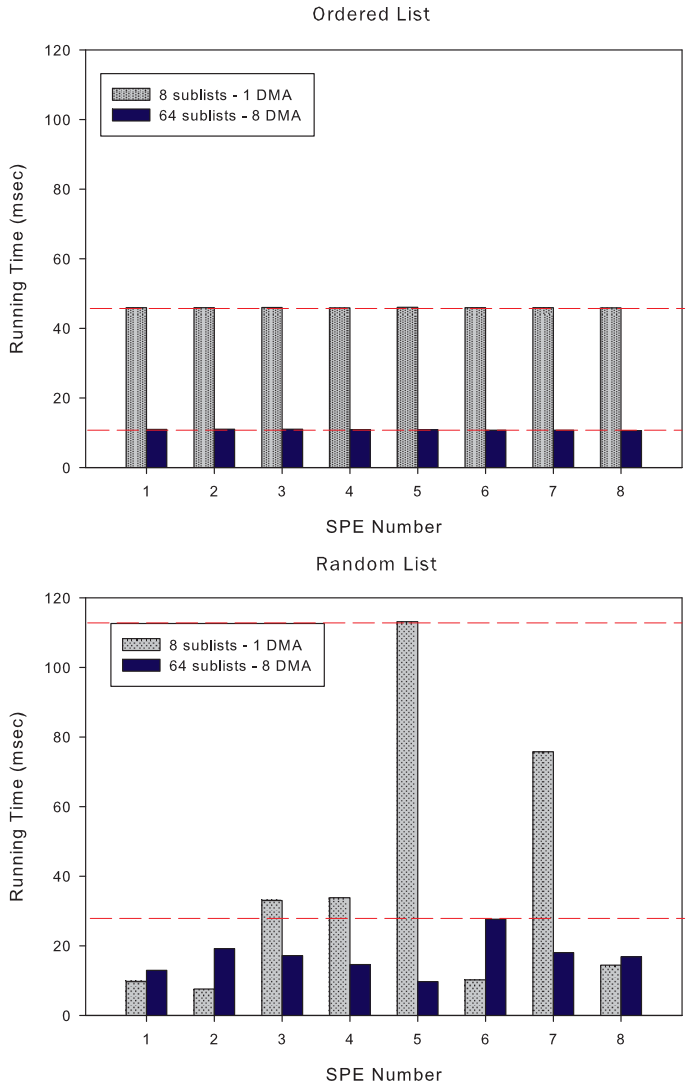

Figure 16: Load Balancing among SPEs for Step 2 of the List ranking algorithm for lists of size 2^{20}. The upper and lower dashed horizontal lines represent the running time of this step of the algorithm for 8 and 64 sublists respectively.

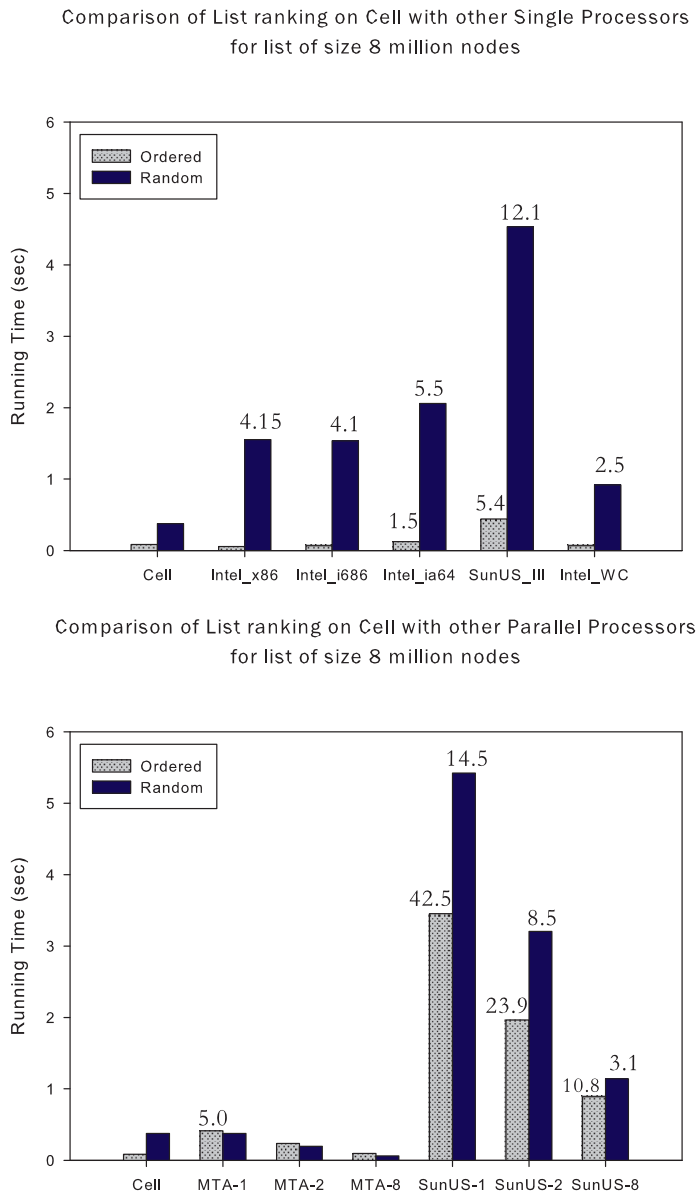

Figure 17: Performance of List ranking on Cell as compared to other single processor and parallel architectures for lists of size 8 million nodes. The speedup of the Cell implementation over the architectures is given above the respectives bars.

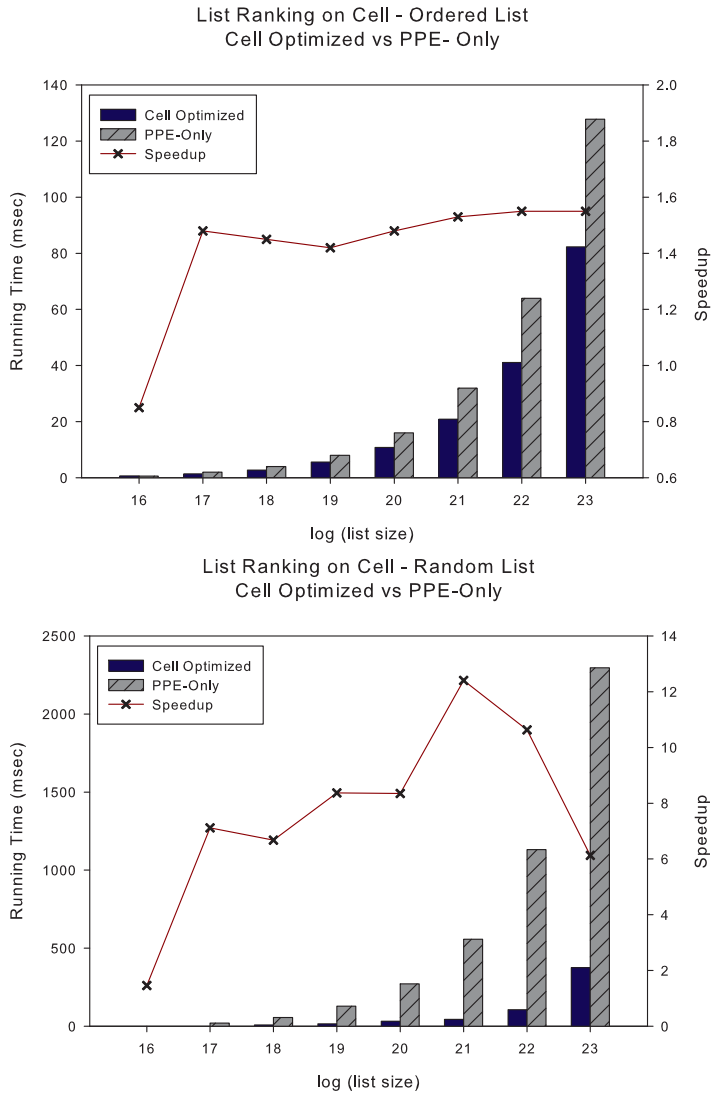

Figure 18: Performance Comparison of sequential implementation on PPE to our parallel implementation of List ranking on Cell for Ordered (Left) and Random (Right) lists

technique can be applied to many irregular algorithms having exhibiting unpredictable memory access patterns. Using this technique, we develop a fast parallel implementation of the list ranking algorithm for the Cell processor and confirm the efficacy of our technique by demonstrating an improvement factor of 4.1 as we tune the DMA parameter. Most importantly we demonstrate an overall speedup of 8.34 of our implementation over an efficient PPE-only sequential implementation. We show substantial speedups by comparing the performance of our list ranking implementation with several single processor and parallel architectures.

We believe that the results presented in this chapter forms a valuable foundation to develop the architectural and algorithmic building blocks of upcoming exascale machines. In future, we will extend the algorithmic design to map other graph exploration kernels and perform scalability studies on system will more sockets.

CHAPTER III

PATTERN RECOGNITION ON MASSIVE STREAMING DATA

Preliminary versions of this chapter were published as:

- F. Petrini, V. Agarwal, D. Pasetto, "SCAMPI: Scalable CAM-based Algorithm for Multiple Packet Inspection", In *Supercomputing (SC '09)*, Portland, OR, November, 2009.
- D. Pasetto, F. Petrini, V. Agarwal, "Tools for Very Fast Regular Expression Matching", *IEEE Computer*, March, 2010 (Cover Feature).

In this chapter we discuss the problem of recognizing patterns from streaming data. Patterns can either be simple strings or more complex regular expressions. Pattern matching is one of the most compute intensive steps in intrusion detection, and also heavily used for image segmentation, matching DNA sequences and handwriting recognition. Network Intrusion Detection Systems (NIDS) are one of the most promising ways to protect systems on the network. Together with firewalls, they provide a first line of defense to attacks that does not require any modification to existing networking and user software. While firewalls limit access based on packet headers, intrusion detection systems go beyond this by identifying attacks that use valid packet headers, by searching both headers and payloads to identify attack signatures. The growing network rates and the large number of signatures that need to be scanned concurrently pose very demanding challenges to algorithmic design and practical implementation. We believe NIDS string searching is a problem that requires the right combination of algorithmic design and abundance of computing resources.

In this chapter we present an innovative string matching algorithm, that achieves speed, space efficiency and resilience in a single framework. More specifically, this chapter provides the following contributions.

- *A new algorithm, that is both space- and time-efficient, and it is very parsimonious in terms of usage of available resources*: The major algorithmic innovations are (1) a compression algorithm that divides the states of the automaton into two parts: a cache of frequently accessed

states and the remaining states that are expressed as a "linear combination" of the cached states; (2) a model of computation based on small CAMs (Content Addressable Memory) that can be efficiently mapped on processor architectures that provide vector extensions and (3) a data layout that enforces data re-use and minimizes memory traffic, with a careful orchestration of the memory requests; the data layout aggregates segments of memory that are likely to be accessed in sequence within the same unit of transfer, such as a cache line.

- *Efficient implementation of the algorithm on the Intel x86 family of processors*: These implementations use a variety of optimization techniques, such as vector instructions and memory pipelining. We are able to obtain a processing rate of 16 Gbits/sec on a dual-socket Intel system under heavy hitting.

- *An extensive performance evaluation that explores the scalability of our algorithm*: Our implementation reaches an impressive rate of 1.2 Gbits/sec per x86 processing core with a dictionary of 3.5 millions of keywords. The analysis shows that an attacker needs to know a large fraction of the dictionary, more than 100k distinct keywords, to degrade the performance of a single core below 0.75 Gbits/sec.

- Perhaps a less measurable, but equally important contribution, is to prove that *by combining a novel memory-centric algorithmic approach with the still untapped potential of general-purpose multi-core processors, it is possible to obtain a performance comparable to special-purpose accelerators*, with a higher degree of flexibility and an attractive price/performance ratio.

3.1 Keyword Scanning and Pattern Recognition

Let the dictionary $K = y_1, y_2, ..., y_k$ be a finite set of strings or keywords and x be the input stream of data. The pattern matching problem is to locate and identify all keywords in K that match at a character of the input stream. Substrings may overlap with one another as well.

3.1.1 Aho-Corasick Algorithm

Our algorithm for scanning keywords is based on the well-known Aho-Corasick algorithm [7], and inherits many of its basic properties. It uses a Non-deterministic Finite Automaton (NFA) as a

computational engine, and can work on strings of arbitrary length. The NFA version of the Aho-Corasick algorithm, which is built out of the keyword tree adding failure transitions, has nice scaling properties. For example it can take advantage of the presence of existing paths when adding new patterns to the keyword tree.

Figure 19(a) describes how to build a simple keyword tree. The strings *testing* and *testcase* share a common prefix and a sequence of states. In the general case, thanks to this property, the number of states in a keyword tree is sub-linear with respect to the total number of characters in the input dictionary.

The keyword tree is extended with a failure function, as shown in Figure 19(b). The resulting keyword tree with failure transitions is often called (incorrectly, because there is no non-determinism in the automaton) the Aho-Corasick Non-deterministic Finite Automaton (NFA). When parsing a character that does not belong to any of the available paths in the keyword tree, we can follow the fail transitions until we identify the longest suffix that is in the keyword tree. This path may, in the general case, require a number of logical steps in the NFA, with at most two steps per input character [7]. The failure dependencies can be resolved for each state, building a fully populated Deterministic Finite Automaton (DFA) in which every character transition can be executed in a single step.

3.1.2 Pros and Cons of the Aho-Corasick Algorithm

Both Aho-Corasick's NFAs and DFAs have pros and cons when they are implemented on a conventional processor architecture or a special purpose accelerator. The NFA is very space efficient –it requires only one extra transition per state in addition to a keyword tree, a marginal size increase in the input. In practice, this makes the NFA space-optimal,[1] but it requires a search across the available transitions in each state to locate the next state, if available, followed by a sequence of searches along the fail chain if the state transition is not available. In a real implementation, each of these steps may require one of more memory accesses, and the run-time overhead may be unacceptable for a practical implementation.

The DFA, on the other hand, minimizes the number of logical steps –the next state transition

[1] The NFA can be further compressed using the techniques described in [140]

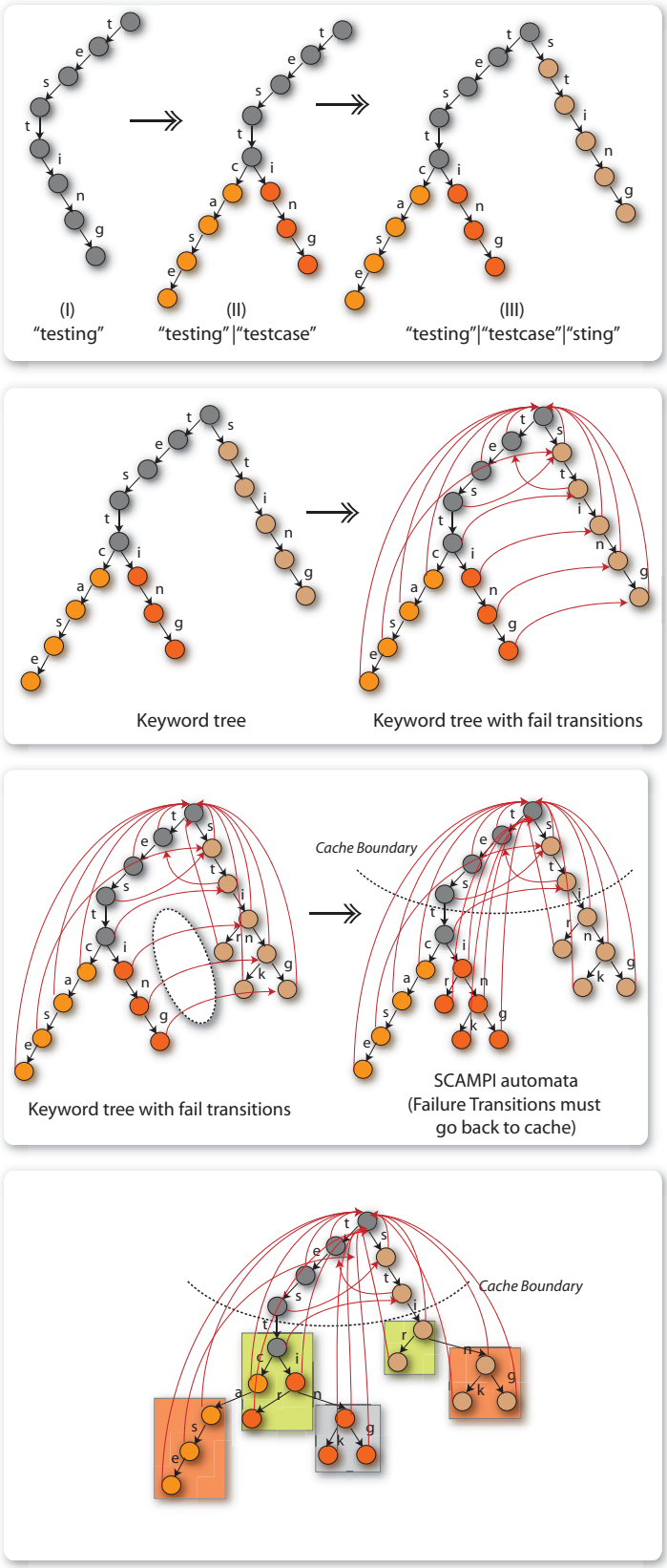

Figure 19: Building the Automata: (a) Building the keyword tree, (b) Adding Failure Transitions, (c) SCAMPI NFA, (d) Locality Enhancing Mapping

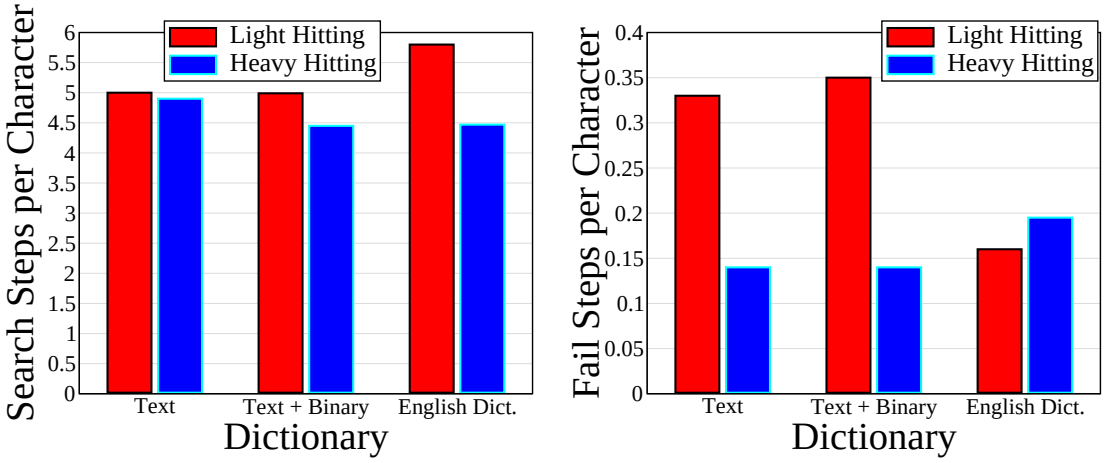

Figure 20: Search steps (a) and fail transitions per character (b) in an Aho-Corasick NFA

can be resolved directly by accessing a memory location with a single memory access and minimal run-time, but each state requires a fully populated array of transitions, one transition for each character in the alphabet (with the ASCII character set and a 32-bit address space, we need 1 KB of memory per state). This space inefficiency can unfortunately lead to a severe run-time overhead. In fact, dictionaries with a few hundreds of thousands of keywords will require several Gbytes of memory, and 64-bit addressability with millions of keywords. For example, the Aho-Corasick implementation in [146] requires more than 2 Gbytes of memory. To intuitively understand the pros and cons of each approach, we have evaluated these aspects using three dictionaries: a textual dictionary with approximately 190k keywords, a binary and textual dictionary of the same size, and a smaller textual dictionary that contains the 20k most common keywords of the English dictionary. We distinguish two cases: heavy-hitting, when the input contains a high percentage of words in the dictionary, and the opposite case, low-hitting. We first consider the behavior of the NFA, counting the number of steps that are required per input character in Figure 20(a) and the number of fail steps per character in Figure 20(b). We can see that, on average, for each character we have to search 5 to 6 entries, and we have a fail transition every three characters in low hitting mode. The NFA search becomes more efficient in heavy hitting mode, because it is following the original transitions of the keyword tree, therefore reducing the number of fail transitions and search at each state (we parse more often states that are further away from the root and have fewer valid transitions). To put this in perspective, to sustain a scanning rate of 1 Gbit/sec, we only have 24 clock cycles per input byte,

on a 3 GHz processor. Searching 5 or 6 entries with a naive sequential algorithm would probably require tens of clock cycles. And a main memory request generated by a fail transitions would also require tens of clock cycles in the most aggressive memory design, and hundreds of clock cycles in a commodity processor. The large memory footprint of the DFA version of the Aho-Corasick algorithm can also lead to a dramatic performance degradation. In Figure 21 we can see that the scanning performance of a DFA of a small dictionary (only 600 keywords) can go from more than 2 Gbits/sec down to 60 Mbits/sec on a current state of art Intel Xeon processor, based on the hitting pattern. With heavy hitting, the access pattern is randomized across the DFA, generating a memory traffic that cannot be handled efficiently by commodity processors. Multi-threading can help, as shown in [146]: using 128 HW threads, the memory latency can be hidden to a point that each thread is able to fully pipeline its memory requests.

Therefore, NFAs and DFAs in their original form are not scalable solutions in both space and speed.

3.1.3 Related Work

String matching is a very popular research subject, with hundreds of publications in the literature. Important contributions to exact string matching are Boyer-Moore [28], Commentz-Walter [47], Karp-Rabin [78], Wu-Manber [152], Muth-Manber [100], and approximate solutions are described in Dharmapurikar et. al [53], and Ramaswamy et. al [118]. Many FPGA based solutions exist for this problem [69, 130], that exploit the large amount of parallelism available on these architectures. However, these solutions are not very scalable with the dictionary size due to the limited amount of fast memory on the device, and are also limited in terms of flexibility and programmability.

Table 4 provides an admittedly cursory overview of recent algorithms/implementations in the literature. We categorize these solutions with respect to the underlying algorithm used, maximum number of patterns tested, space requirements and memory footprint, expected scalability to larger dictionaries, performance for dictionaries considered as reported in the publication, architecture on which the algorithm has been tested for, and resilience under attack. The attack in this analysis refers to the degradation in network performance with inputs representing increasing knowledge of the dictionary used for pattern recognition. The goal is to provide an intuitive and visual description

of existing algorithms and to expose the positive sides, and the potential weaknesses that limit their use in a practical scenario.

Villa et. al [146] present an implementation of a DFA based on the Aho-Corasick [7] algorithm, with optimizations specific to the Cray XMT multithreaded architecture [51]. The algorithm reaches a performance of 28 Gbits/sec with 190k patterns, however the memory footprint of the algorithm is $O(m)$, where m is the number of characters in the dictionary, with a large order constant of one KB per state, as discussed in more detail later in Section 3.1.1. Therefore, the scalability is limited when we increase the dictionary size. This solution is also prone to attacks, where an input with a recurring small subset of patterns from the dictionary would result in repeated accesses to the same area of memory and create hotspots. The authors address this problem by replicating part of the automaton, as already done in [147], further exacerbating the space problem.

Liu et. al [90] present an implementation that is based on Bloom filters supported by an exact matching engine (based on a hash table), on the IBM Cell/B.E. processor with 16 Synergistic Processing Elements (SPEs). Their algorithm attains a performance of 3-17 Gbits/sec up to 190k patterns, with varying frequency of hitting in network data. The exact matching engine requires $O(m)$ space. The intuition behind this algorithm is that the system is generally working in skipping mode and therefore the Bloom filters can quickly parse the input stream. The scalability of this implementation is limited by the small local store on the SPE of the Cell processor; also the quality of the Bloom filter decreases, creating many false positives as the dictionary size increases. A system based on this implementation is also prone to attacks: a recurring input with a few dictionary patterns could frequently initiate both the filtering and matching stages of the algorithm.

Lunteren [142] presents a BFSM based pattern matching algorithm for hardware accelerators (ASIC, FPGA). This algorithm creates compressed cluster representations of the transition rules based on the input dictionary. The space requirement is quadratic with respect to the number of patterns in the worst case, and super-linear on average, thus the scalability is severely limited with increasing dictionary size. For a dictionary with 2k patterns, a performance of 20 Gbits/sec is estimated on the ASIC technology. The hardware implementation is expected to be resilient to attacks, however a cache based implementation of the algorithm suffers from poor memory locality.

Salmela et al. [121] present an implementation based on q-Gram filtering and Rabin-Karp [78],

combined with binary search and two-level hashing for exact matching. The memory footprint of the algorithm is low for the q-Gram filtering phase, where the total memory depends on the value of q, with larger q resulting in higher memory requirements. However, the exact matching algorithm requires $O(m + n)$ space, with respect to number of total characters (m) and number of patterns (n). This solution is limited in terms of scalability due to the increasing number of false positives with larger dictionary sizes. Also, the system is prone to network attacks, as an input with recurring matching patterns can always result in a match at the filtering stage, resulting in the frequent invocation of the exact matching engine.

Weinsberg et. al [148] present a Ternary Content Addressable Memory (TCAM) based specialized hardware approach. The entire dictionary is divided into TCAM entries, with larger patterns spanning multiple entries (depending on the TCAM width). Every entry has a *shift* value associated with it, that specifies the amount of input to skip if there is a match. For *shift* value 0, the rest of the pattern is matched with the input using more TCAM lookups. The algorithm utilizes w^2r bytes of space, where w is the TCAM width, and $r = \Sigma\lceil\frac{m_i}{w}\rceil$, m_i being the ith pattern length. On a 26k ClamAV pattern set, the algorithm attains a simulated performance of 31-52 Gbits/sec, under light hitting. The system is prone to attacks, with high variability in performance when the input forces the lookup of zero shifts TCAMs all the time. This degrades the performance to 2 Gbits/sec in the worst case.

Vasiliadis et. al [143] present an implementation of the Aho-Corasick [7] on NVIDIA G80 GPU. This implementation requires a full state transition table, which accounts for a huge memory footprint, limiting scalability to large dictionaries. On a 4k pattern set from Snort the implementation achieves a performance of 1.4 Gbits/sec, with a peak of 2.3 Gbits/sec. Under network attack a system based on this implementation will require many of expensive memory transfers, with a serious performance penalty.

Erdogan et. al [55] extend the ClamAV open source software to include Bloom filtering prior to the exact matching. The Bloom filtering working set can fit in cache, however for large dictionaries it can generate many false positives. For a dictionary with 112k patterns the implementation achieves a performance of 13-120 Mbits/sec on the AMD Athlon XP 2000+. This implementation is also not resilient to attacks, as an input with a recurring dictionary pattern can result in frequent filter

matches.

3.2 Scalable CAM-based Algorithm for Multiple Packet Inspection

3.2.1 Intuition

In this section we present our Scalable CAM-based Algorithm for Multiple Packet Inspection (SCAMPI). SCAMPI tries to achieve the best of both worlds –a space efficient automaton that requires only a minimal, constant overhead for each search step. Our algorithm is based on three main ideas.

1. A compression algorithm that divides the states of the automaton into two groups: a small group of frequently accessed states, which we call *SCAMPI cache* or simply *cache*, that acts as generating base for all the states that are in the second group, the *uncached* states. The goal is to resolve each state transition with at most a single main memory access.

2. A computational building block that can be executed in constant time, that is based on a simple CAM search. Rather than having a large, monolithic CAM, we rely on a collection of small CAMs. The SCAMPI computation is almost entirely based on this building block.

3. An arsenal on locality-enhancing techniques to allocate these small CAMs in a way that is memory efficient and minimizes the number of main memory accesses.

Figure 19(c) shows how the SCAMPI automaton is built starting from the Aho-Corasick's NFA. Uncached states expand their failure chain until they get to a cached state. The resulting automaton has a hybrid structure, with the cached states that borrow the topology of the NFA and the uncached states that are enhanced with all possible suffix information and fail transitions that are not contained in cache. Therefore, the next state transition of each uncached state can be resolved either within the state transitions or a cached fail state, making each uncached state *self-contained* in terms of main memory accesses, eliminating the fail steps for uncached states, as shown in Figure 19(d).

For example, in the specific case of Figure 22, the parsing of the string *testasting*, requires 13 steps, counting the fail transitions, with a string of 10 characters, and only two main memory requests, the transitions from state 4 to 5, and from state 10 to 11. All the other transitions can be

Table 4: Related work in keyword scanning

Reference	Base engine/algorithm	Dictionary size	Space utilization	Scalability	Performance	Base architecture	Resilience to attacks
Villa, Miranda, Maschhoff [146]	Aho-Corasick (DFA) with optimizations on Cray XMT	190k	$O(m)$, m: number of characters in dictionary (Full State Transition Table) see Section 3.1.1	Limited by memory on system	$\approx$ 28 Gbits/sec	Cray XMT (Eldorado) 128 HW threads per processor	Knowledge of a few long keywords in the dictionary can result in access of a very small subset of states, resulting in memory hotspots.
Liu, Liang, Zou, Joseph [90]	Bloom filter with exact matching engine	190k	Low for Bloom filter, High for exact matching engine	No, Hash table size limited by small local store and number of false positives increase with number of patterns.	$\approx$ 3-17 Gbits/sec	IBM Cell/B.E. (16 SPEs)	No, high hitting input cases always pass from Bloom filter to exact matching engine.
Lunteren [142]	B-FSM based pattern matching (Transition rules)	$\approx$2k	Low for small dictionaries, Very High for large dictionaries (Super-linear with worst case Quadratic, w.r.t. number of patterns)	No, because of large space utilization	$\approx$20 Gbits/sec (estimated)	ASIC technology	Yes for hardware implementation), No for Cache based architectures (poor locality)
Salmela, Tarhio, Kytöjoki [121]	q-Grams (filtering with exact matching)	100k	Low for filtering phase (depends on q value), O(n) for exact matching as hash value for each pattern, n: number of patterns	No, large number false positives	N/A	N/A	No, limited knowledge of dictionary results in large false positives, binary search in exact matching engine is inefficient.
Weinsberg, Tzur-David, Dolev, Anker [148]	Ternary CAM (TCAM) based algorithm (Mostly shifts for light hitting)	$\approx$ 26k (ClamAV)	Low, memory utilization = $w * w * r$, w: TCAM width, $r = \Sigma\lceil \frac{m_i}{w} \rceil$, m_i: ith pattern length	Depends on available memory, for small TCAM width many associations per entry compromises performance, larger TCAM width increases memory.	31-52 Gbits/sec (ClamAV), 12.35 Gbits/sec (Snort)	Simulated results	No, little knowledge of dictionary can result in zero shift at every byte.
Vasiliadis, Antonatos, Polychronakis, Markatos, Ioannidis [143]	Full Aho-Corasick on NVidia G80	4k (Snort)	High, Full State Transition Table	No, performance of Aho-Corasick for large dictionaries would be limited by available memory on GPU.	2.3 Gbits/sec (peak), 1.4 Gbits/sec (Snort)	NVidia G80	Lots of memory transfers under attack, low performance.
Erdogan, Cao [55]	Bloom filtering prior to ClamAV	112k	Bloom filtering can fit in cache	No, potentially large number of false positives with larger dictionaries	13-120 Mbits/s	AMD Athlon XP 2000+	No, little knowledge of dictionary can results in false positive at every byte.

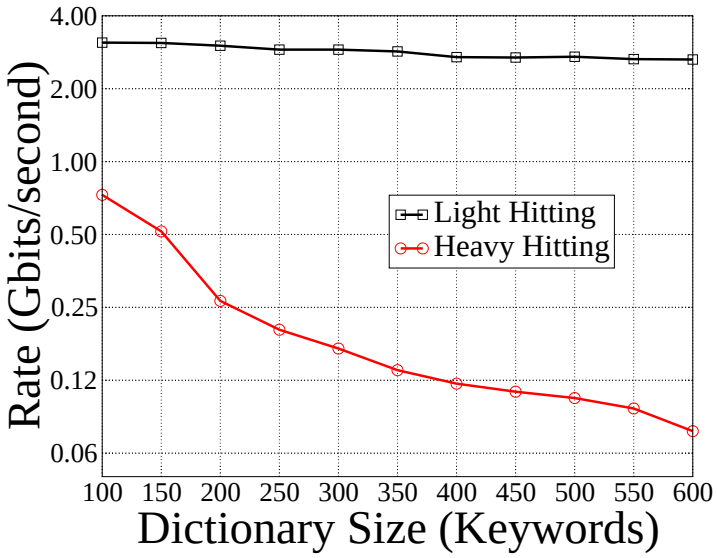

Figure 21: Performance of a small Aho-Corasick DFA with different input sequences and hitting rates

resolved with cached information, thanks to the data layout that maps the information of states 5-6 and 11-12-13 in single units of memory transfer (e.g., a cache line or a DMA).

3.2.2 Analysis using Roofline Model

We believe that for the recent past and the foreseeable future, off-chip memory bandwidth will often be the constraining resource. No matter whether the architecture that we are considering has limited amount of parallelism, such as a commodity multi-core processor, or a rich set of computational engines, such as a GPGPU or an FPGA, applications that display irregular access patterns over a large data sets are very likely to be bottlenecked by the main memory performance.

This line of thought has been elegantly summarized by Williams et at. in the *Roofline* model [151]. This model relates processor performance to off-chip memory traffic, the traffic between the caches and and the memory, rather than between the processor and the caches, combining together computational performance and memory performance in a two-dimensional graph.

Following the footsteps of the Roofline model, SCAMPI is designed to minimize the memory traffic, with a careful orchestration of the memory requests, and to enhance date re-use with a compile-time data layout that aggregates segments of memory that are likely to be accessed in sequence within the same unit of transfer, such as a cache line or a DMA. Figure 23 correlates the main performance components of the SCAMPI algorithm (the two computational components,

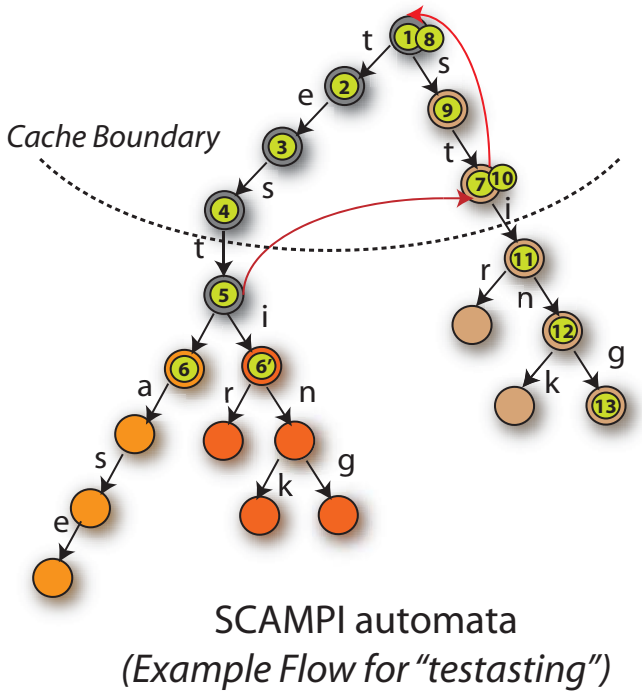

Figure 22: In a SCAMPI automaton each *uncached* state can resolve a state transition either within its CAM or through a sequence of cached states. Therefore, each state transition can be resolved with a single main memory access.

the CAM search and the full state search, with the degree of memory pipelining, represented by multiple automata mapped onto the same processing core) in a single bi-dimensional graph.

3.2.3 Basic Performance of this Algorithm

Figure 24(a) shows how the size of the resulting SCAMPI automaton compares with the keyword tree, the NFA and the full DFA for a collection of dictionaries of incremental size that are obtained from the text and binary dictionary that we use in most of our experiments.

By considering the Aho-Corasick NFA as a practical lower bound for the SCAMPI automaton, we can see in Figure 24(b) that the optimality ratio –the ratio between the SCAMPI and the NFA sizes, is only a factor of two. A size increase is due to the alignment and padding costs, to align each CAM to 16-bytes (to enable SIMD instructions) and group CAMs in a cache-friendly layout. In most architectures, unaligned loads and stores experience increased latencies, and it is usually better to have a slightly larger automaton rather than the extra overhead at run-time. The essence of SCAMPI is best summarized by Figure 25. Figure 25(a) shows the percentage of main memory

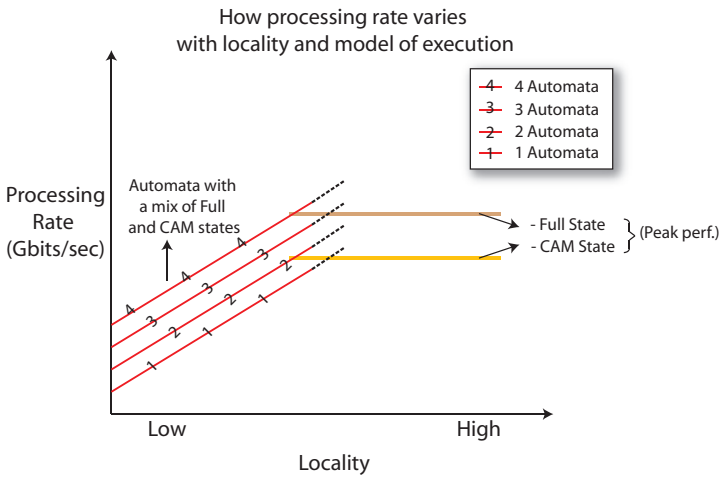

Figure 23: The Roofline model applied to SCAMPI

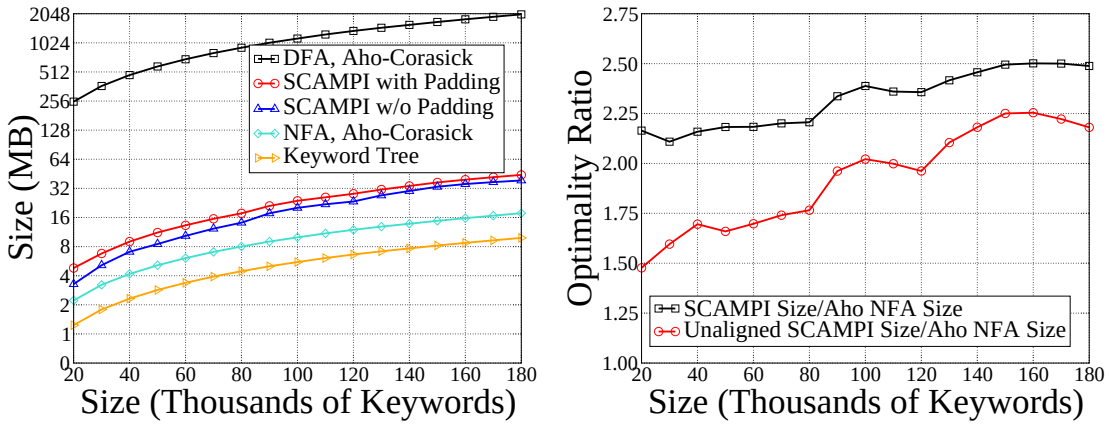

Figure 24: SCAMPI space requirement: (a) Automata size, (b) Optimality ratio

requests issued by the SCAMPI automaton. With a unit of transfer of 64 bytes (the cache line size in x86 processors), less than 5% of the state transitions require a main memory access: in more than 95% of the cases, the state transition information can be satisfied by the SCAMPI cache or by a recently issued memory request. Figure 25(b) can be compared directly with Figure 20(b), the failure transitions of the Aho-Corasick NFA: in all cases the failure steps are reduced by an order of magnitude, to a failure step every 30 characters. For practical purposes, the SCAMPI NFA can resolve each transition in a constant number of steps and it is very likely to find the state information in cache.

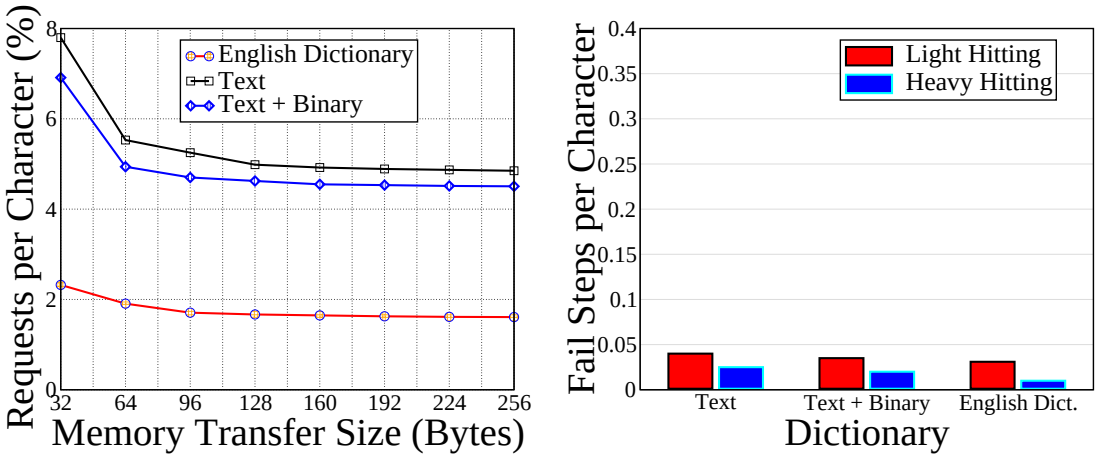

Figure 25: SCAMPI NFA analysis: (a) Percentile of memory requests per input character, (b) Fail transitions per input character

3.2.4 The SCAMPI CAM

The logical organization of each state of the SCAMPI automaton is a collection of entries that must be searched to find the next state (or the failure state) from the given input character. It would be ideal to search all the available transitions in parallel, as done in the CAM shown in Figure 26. The availability of a CAM at each state would allow a search in constant time, reaching the same speed of the Aho-Corasick DFA.

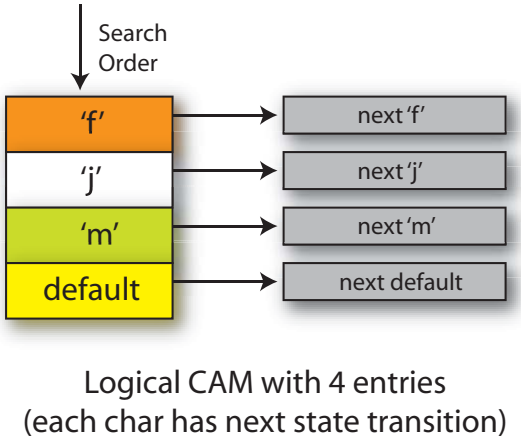

Figure 26: Logical CAM representation

The whole SCAMPI run-time relies on this assumption: a CAM can be easily and efficiently implemented both in software and in hardware. More specifically, the availability of SIMD/vector instructions in commodity multicore processors makes the implementation of software CAMs a

very efficient solution. In Figure 27 we can see how a software CAM can be implemented using vector instructions. The input character can be replicated on an input vector that is compared with a group of reference vectors, two in our case, that represent the CAM entries. Using a bit gather operation, a *vector reduction* operator, we can squeeze the result of the matching in a single scalar register, and with the count lead zeros operation we can identify whether there is a match in the CAM and its location in the CAM array. This index can be used to locate the next state with a single memory access. While this is not as simple as a single memory reference, it can be executed in a handful of clock cycles with processors that are equipped with vector units, enabling peak scanning rates of several Gbits/sec per processing core.

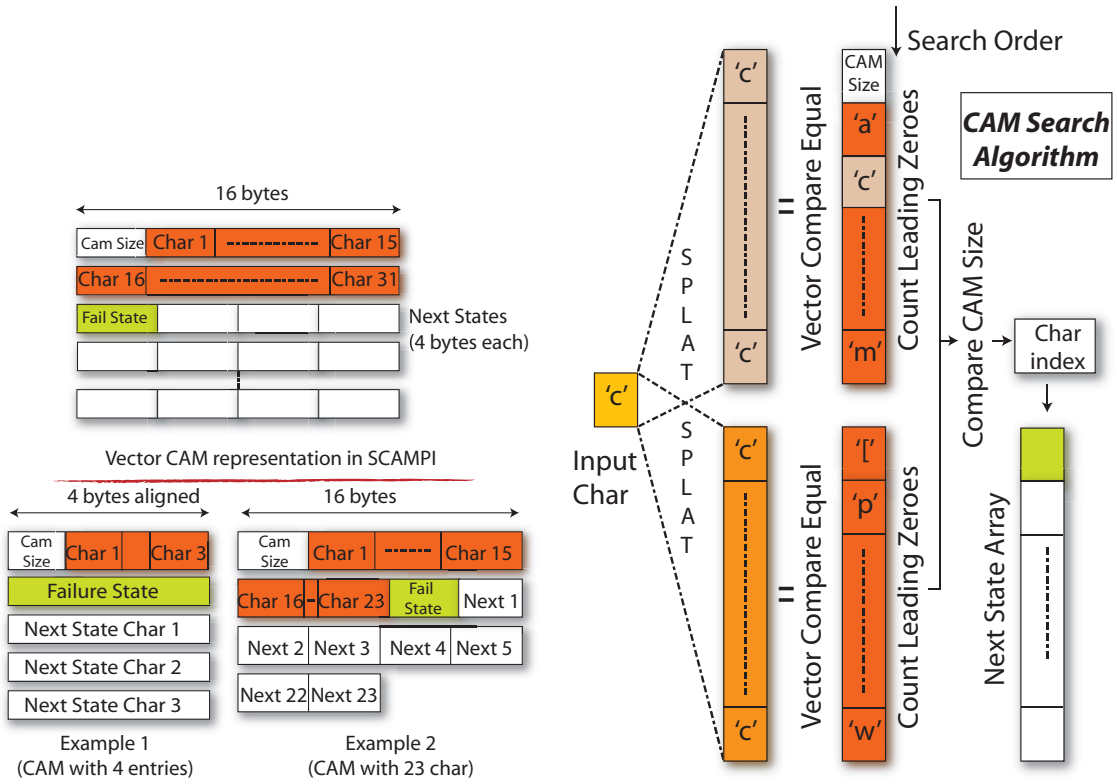

Figure 27: A SCAMPI CAM implemented with vector intrinsics: (a) SCAMPI data layout using vectorized CAM, (b) CAM vectorized search algorithm

3.2.5 SCAMPI Compiler

Figure 28 provides an overview of the various components of SCAMPI. The central part of our algorithm is the pattern compiler that takes a dictionary of unique patterns and an optional training input and generates a SCAMPI automaton and various support data structures, including the match

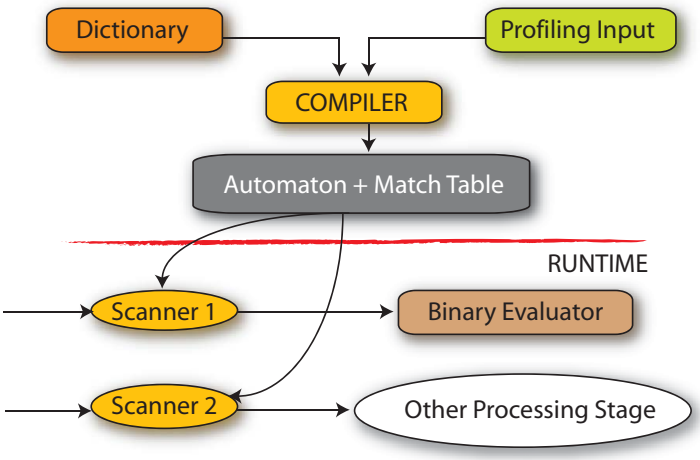

Figure 28: SCAMPI framework: The two major algorithmic components of SCAMPI are (1) pattern compiler, (2) run-time algorithm

table, that are utilized at run-time by the keyword scanners to parse the input data streams. The main compilation steps are the following.

1. **Build NFA**: this stage is similar to the native Aho-Corasick algorithm and builds a keyword tree that is later enhanced with fail transitions.

2. **Profile NFA**: this an optional stage of the compilation that, given a training input, identifies the states that are more likely to be visited, and for each state the transitions that are more likely during the keyword scanning. Hot states are natural candidates for a cache allocation, while hot transitions can lead to an optimized allocation of the CAM layout of each state.

3. **Build Profiled Cache**: using the hints of the previous stage (if available), we select a few states that are frequently used and have topological properties (for example they are close to the root of the NFA, or are the root of a common prefix) that suggest a cache placement.

4. **Build BFS Cache**: in the absence of profiling information, the NFA cache is built using a Breadth First Search (BFS).

5. **Build Main/Build Chains**: this is the central part of the algorithm that builds all the CAMs of the uncached states. In this phase we group the states in *linear chains*, collections of states that are likely to be accessed in a short temporal sequence, to increase the run-time performance by increasing memory reuse.

6. **Map CAMs**: in this stage we map the logical state CAMs to physical addresses, performing memory alignment (all the CAMs start at a 16-byte boundary) and padding.

7. **Write NFA**: the NFA and other relevant data structures are written in a file using a network-independent format.

The most computationally intensive phase is the *Build Main/Build Chains* one, where we build the CAMs for each uncached state and aggregate the CAMs in units of data transfers. The complexity of this phase is linear in the number of states, and each state is visited only once. The SCAMPI compiler is remarkably fast, thanks to its single-pass organization that has linear complexity with the number of states. The typical compile time for a dictionary of 200k keywords, like the one commonly used in our tests, is only 30 seconds on a current state of the art Intel x86 CPU. Moreover, the compiler has a reduced memory footprint, and can be executed on a 32-bit address space.

3.3 Experimental Results

3.3.1 Basic Performance of SCAMPI

To run SCAMPI, we use the Intel Xeon E5472 processor running at 3.0 GHz, with 4 cores per socket, and 6MB cache. We analyze basic performance using a collection of algorithms, that can be roughly characterized by the degree of vectorization and the usage of the SSE intrinsics. We can choose between algorithms that have a moderate level of vectorization, to algorithms that can use special intrinsics, such as those provided by the SSE 4.2, that can directly execute a CAM search. In this chapter we have used one of the least aggressive vectorizations, which is based on the SSE2 intrinsics, in order to have a more portable solution.

The performance of the scanning algorithm described in Figure 29(a) is strongly influenced by the hit rate and the memory locality, with performance ranging from 2.1 to 0.5 Gbits/sec. This figure compares performance of SCAMPI NFA for a large dictionary under various network/input (1) when the input results in looping over full states, (2) when the input results in hitting mainly CAM states, (3) an input that results in very heavy hitting, (4) feeding the source dictionary as the input. Even in the worst case when the dictionary is fed to itself, SCAMPI manages to provide an excellent level of performance. Finally, the scaling of all algorithms is shown in Figure 29(b). The

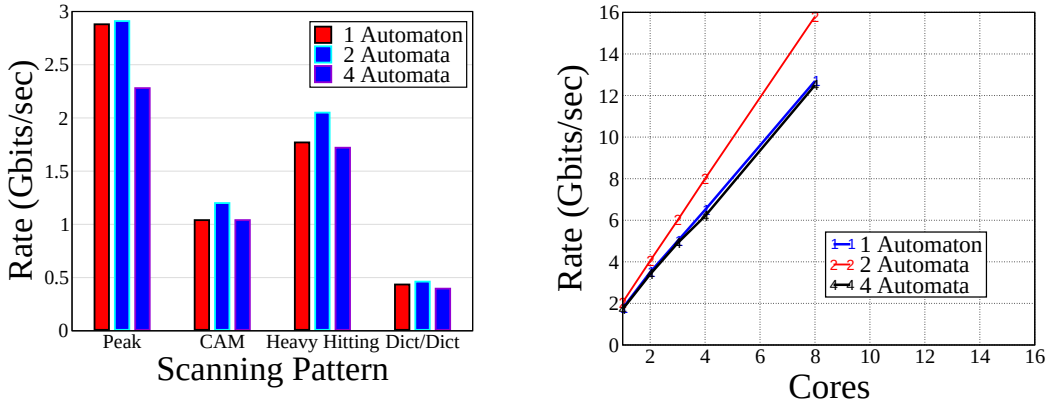

Figure 29: SCAMPI Performance: (a) Performance sensitivity to hitting rate and memory locality, (b) Scaling Performance

best performance is obtained with 2 automata. The optimal number is typically a function of the degree of pipelining of the memory interface.

3.3.2 Scaling Analysis

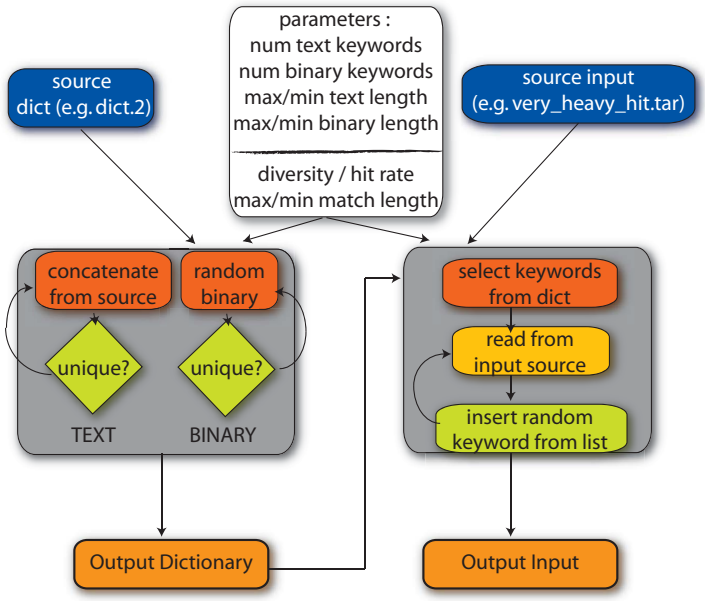

Figure 30: Algorithm for Dictionary and Input generation

In this section we discuss and analyze the scalability of SCAMPI with increasing dictionary sizes and the resilience of the algorithm under extreme conditions. We developed a framework to generate dictionaries of different sizes and characteristics, starting from a user provided dictionary

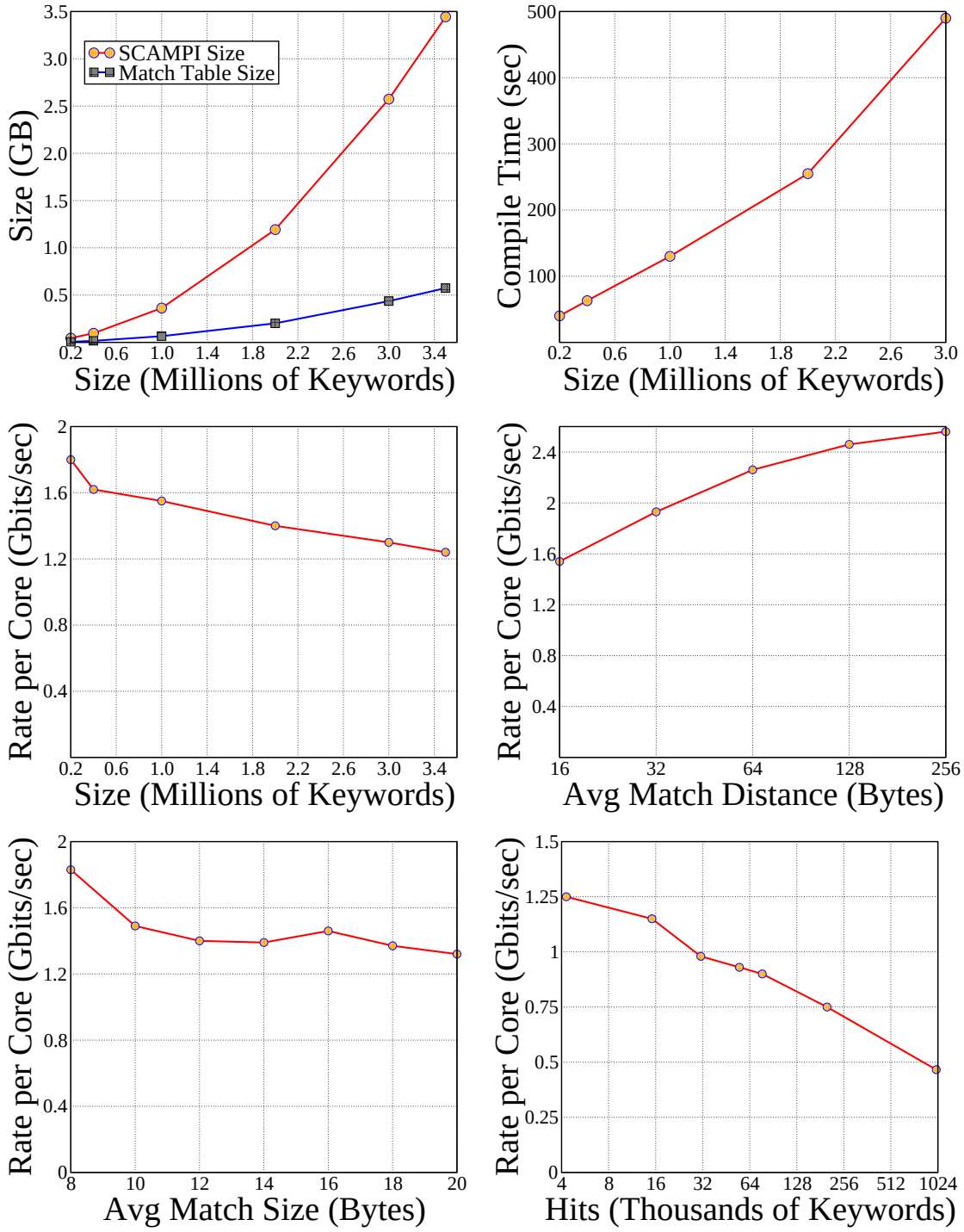

Figure 31: SCAMPI scalability and performance sensitivity: (a) SCAMPI NFA size as a function of the dictionary size, (b) Compile time, (c) Dictionary Scaling, (d) Frequency of the matches, (e) Match Size, (f) Number of Unique Hits

(which in our experiments is the text and binary dictionary of the previous experiments).

Figure 30 illustrates the control flow of our dictionary and input generation framework. If the number of text patterns generated by this framework is less than those contained in the source dictionary, it randomly chooses a subset of the source patterns. If the patterns are generated more than once, it concatenates them and performs a post-processing to match the various input parameters such as the min/max pattern length. For each generated pattern it checks for uniqueness with respect to the previously generated patterns. If the pattern is not unique, it randomly changes a subset of the characters in the generated pattern. For binary patterns, it generates a sequence of random bytes based on the input specification and checks for uniqueness with respect to all previously generated patterns. Again, if the pattern is not unique, it randomly changes a subset of bytes until the pattern is unique.

With increasing dictionary size it becomes important to analyze the space requirements of the algorithm. With the limited memory available on processors and large memory access latencies this becomes a critical factor for performance scalability and the practicality of an algorithm in production systems. Figure 31(a) gives the memory requirements of SCAMPI with increasing dictionary size. The dictionaries used for this analysis contain an equal mix of both text and binary patterns, with average pattern length of about 16. The x-axis in the graph gives the size of the dictionary and the y-axis gives SCAMPI memory utilization in Gbytes. The *SCAMPI size* plots the size of the SCAMPI automata and the *Match Table Size* gives the size of the match table containing information on all the matched patterns for each final state. We observe that for dictionaries with less than 200k keywords, the space requirement of SCAMPI is well below 100 Mbytes, and even for very large dictionaries containing 3.5 million patterns the space requirement is of the order of a few Gbytes.

Figure 31(b) gives the compile time for different dictionaries generated using our generation framework, using the reference text plus binary with 190k keywords. The x-axis gives the size of the dictionary varying from 200k patterns to 3 millions of patterns. The y-axis gives the compile time in seconds. As described in the previous sections, the compiling of the dictionary refers to the software compile that generate the automata, which is then used at run-time to detect the matches. For most frequently used dictionary sizes the compiling time is well under a minute. Even for

dictionaries containing one million patterns, the compile time is of the order of two minutes.

Next, we analyze the running time of our algorithm for a given input with increasing dictionary size. Figure 31(c) gives the processing rate per core on an Intel Xeon E5472 processor (3 GHz). We used a reference input file with high frequency of hitting for all these tests. The dictionaries contain an equal mix of text and binary patterns. The x-axis in the figure gives the size of the dictionary and the y-axis gives the processing rate per core in Gbits/sec. We observe here that the processing rate falls from 1.8 Gbits/sec as we increase the dictionary size, but stays over 1.2 Gbits/sec even for a dictionary with 3.5 million patterns.

Figure 31(d) gives the processing rate per core on the same processor as above, for a dictionary with one million keywords, with increasing frequency of keyword hit in the input. We use an input file that contains an average pattern length of 8 bytes of the matched keywords. The x-axis gives the average distance at which a keyword was inserted using our input generation program, and the y-axis gives the processing rate per core. We observe that the performance stays above 1.6 Gbits/sec even for a very high matching frequency. We also observe here that the performance of our implementation drops by only a very small factor even with a very high hit rate, thus making it resilient to attacks.

Along with the frequency of a match, it is also very important to analyze the performance variation with respect to the size of the match. Figure 31(e) shows the performance of SCAMPI under varying length of keyword matches. We used a dictionary with 1 million keywords with an equal mix of both text and binary patterns. A number of inputs were generated for this performance analysis by inserting keywords with increasing average pattern length. We observe that the performance of our implementation is within 1.3-1.8 Gbits/sec, with average match size increasing from 8 to 20. SCAMPI speculatively caches the next states that would be visited when the next state access goes outside the cache. Thus, long pattern matches cause only a small increase to the number of main memory accesses, making the performance resilient to match length.

Here, we introduce a very important factor, *diversity*, that can affect the performance of a system and is a critical factor for analyzing network attacks. The *diversity* of an input is the number of different keyword matches in a given unit of time. Increasing the diversity of an input requires extensive knowledge of the dictionary, and can only happen under rare circumstances. Figure 31(f)

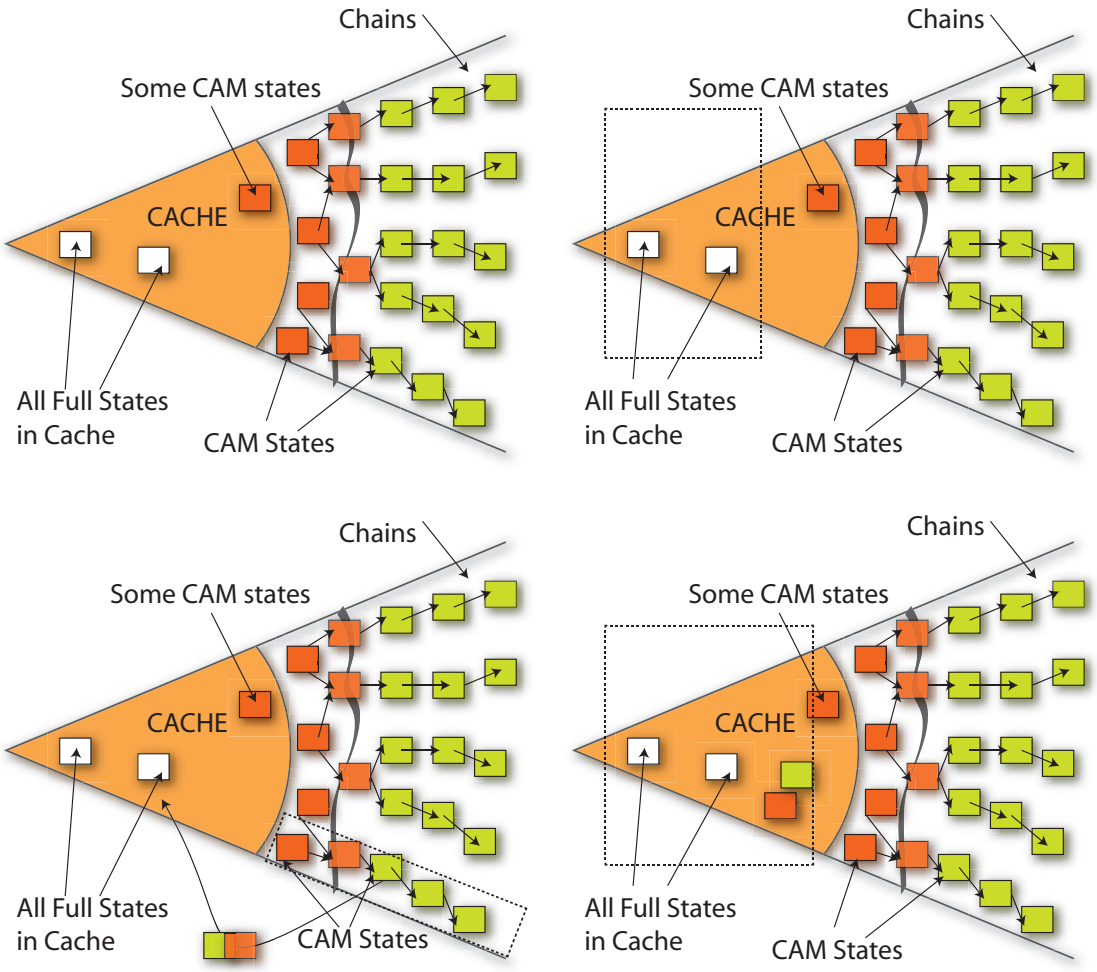

Figure 32: What happens when SCAMPI is attacked: (a) SCAMPI automata structure, (b) High frequency attack using small dictionary words, the states remain cached, (c) High frequency attack using long dictionary words, (d) These states are now cached

analyzes the performance of SCAMPI under a network attack where we increase the diversity of the input. While generating the input we randomly select a pattern from a subset of patterns, at each insertion point. To increase the diversity of the input, we increase the size of this subset. We maintain a high hit frequency in the input to simulate a real network attack. The x-axis gives the number of unique hits (in terms of the size of the subset), and the y-axis gives the processing rate per core. We have used a dictionary of 1 million patterns with an equal mix of both binary and text keywords. We notice that our system is extremely resilient to attacks, with performance staying above 0.75 Gbits/sec up to a diversity of 200k keywords. The worst case scenario is when attacker has a complete knowledge of the reference dictionary, thus the extreme point of the graph is where

the same dictionary of 1 million keywords is used as the input. It is important to note that SCAMPI achieves a performance of 466 Mbits/sec per core even under such attacks.

Finally, Figure 32 illustrates the behavior of SCAMPI under a network attack. Figure 32(a) gives the structure of the SCAMPI automata. The automata consists of two parts, the states that are cached and are frequently hit; and the states that are outside the cache. With SCAMPI, the next state transitions of states outside cache tend to have a chained behavior and a failure transition results in an access to a cached state. Figure 32(b) shows the behavior of the system during a network attack under heavy hitting using small dictionary. In this case the states that are already in the cache are hit, thus there is no performance penalty. A network attack that uses long dictionary words, as illustrated in Figure 32(c), results in accessing states that are outside the cache. However, thanks to the locality-enhancing data layout, the next state transitions are likely to be in cache.

3.4 Summary

There is no doubt that the growing network traffic is exposing serious bottlenecks of the current network intrusion detection systems. This problem is aggravated by the large number of signatures that need to be scanned concurrently and at line rate. In this chapter we present SCAMPI, an innovative algorithmic solution that is able to achieve space efficiency and high speed with very large dictionaries, even when the system is under attack. SCAMPI is composed by a pattern compiler that takes a dictionary and optional training input to generate an automaton, other optimized data structures, and an efficient run-time system that takes this automaton to parse input data streams. We design a novel automata compression algorithm, a model of computation based on small software CAMs and a unique data layout that forces data re-use and minimizes memory traffic. Our optimized implementations on a range of high-end Intel x86 processors, achieve a processing rate of 16 Gbit/sec in a dual-socket configuration under heavy hitting. We also present an extensive evaluation that explores the scalability of SCAMPI to millions of keywords, the performance obtained with various input and dictionary characteristics, and demonstrated the resilience of the framework under network attacks.

CHAPTER IV

DATA ANALYSIS ON STREAMING FINANCIAL MARKET FEEDS

Preliminary versions of this chapter were published in:

- V. Agarwal, D.A. Bader, L. Dan, L.-K. Liu, D. Pasetto, M. Perrone and F. Petrini, "Faster FAST: Multicore Acceleration of Streaming Financial Data", *Computer Science - Research and Development*, Springer, 23(3):249-257, 2009.

- V. Agarwal, L.-K. Liu, and D.A. Bader, "Financial Modeling on the Cell Broadband Engine," 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), Miami, FL, April, 2008.

- D.A. Bader, A. Chandramowlishwaran, and V. Agarwal, "On the Design of Fast Pseudo-Random Number Generators for the Cell Broadband Engine and an Application to Risk Analysis," The 37th International Conference on Parallel Processing (ICPP 2008), Portland, OR, September, 2008.

A typical market data processing system consists of several functional units that receive data from external sources (such as exchanges), publish financial data of interest to their subscribers (such as traders at workstations), and route trade information to various exchanges or other venues. A high-level design of a market data processing system or *ticker plant* is sketched in Figure 33. Examples of functional units include feed handlers, services management (such as permission, resolution, arbitration, normalization, etc.), value-added, trading system, analytics, data access, and client distribution. The feed handler is the component that directly interacts with the feed sources for handling real time data streams, in either compressed or uncompressed format, and decodes them converting the data streams from source-specific format into an internal format, a process often called *data normalization*. According to the message structure in each data feed, the handler processes each field value with a specified operation, fills in the missing data with value and state of its cached records, and maps it to the format used by the system.

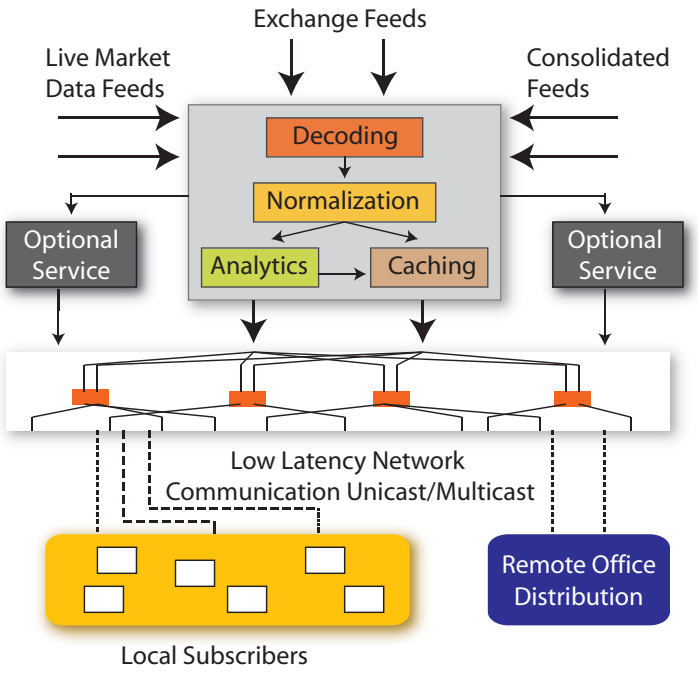

Figure 33: High-Level Overview of a Ticker Plant

Financial data feeds in global options and equity markets are expected to produce an average of more than 128 billion messages per day by 2010, rising from an average of more than 7 billion messages a day in 2007 [138]. In the options market, the OPRA (Option Pricing Reporting Authority) consolidated feed for all US derivatives business currently represents a very significant portion of market data in the national market system. Figure 34 shows that the OPRA market data rates, on a one-minute basis, have dramatically increased over the course of the past 4 years, approaching 1 million messages per second. The traffic projection for OPRA alone is expected to reach an average of more than 14 billion messages a day in 2010. It is important to note that alongwith the ability to process a high throughput of data, low latency is critical to the success of the market data processing system, especially with the increasing competition among financial institutions as well as diminishing profit margin. How fast a trading system can respond to the market will determine who wins and who loses, even a few milliseconds difference in latency is enough to make the difference. Obtaining low latency while maintaining high throughput in processing market data feeds exposes further challenges in algorithmic design.

This chapter explores multiple avenues to deal with Option Price Reporting Authority (OPRA) market data feed decoding and normalization on commodity multicore platforms, and describes a

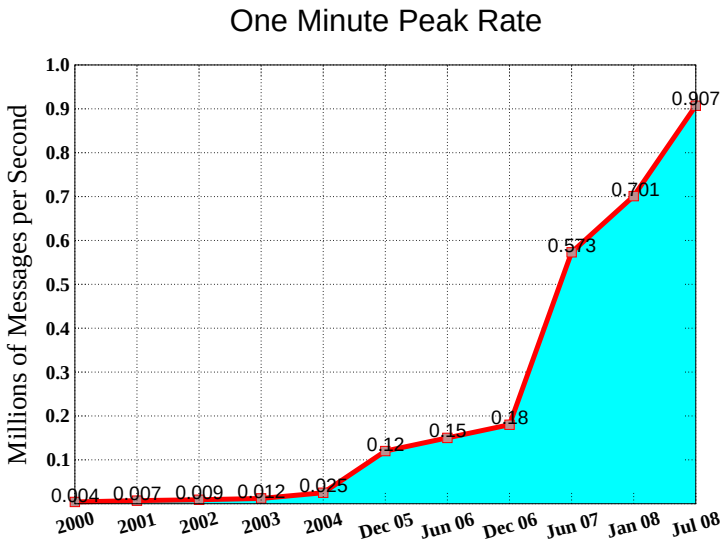

Figure 34: OPRA market peak data rates

novel solution that is able to capture the essence of the OPRA protocol with a high-level description language. We believe our solution can also be easily extended to other market data feeds by modifying the high level grammar using the protocol specification. With the increase in volume of market data feeds and financial instruments, high performance computing is increasingly becoming critical for financial markets where analysts seek to accelerate complex optimizations such as pricing engines to maintain a competitive edge. In this chapter we present solutions and optimization techniques to accelerate the Option pricing financial analytics workload using Monte Carlo simulation on the IBM Cell B.E..

More specifically, this chapter provides the following contributions:

- *The design of a high-speed hand-optimized OPRA decoder for multi-core processors*. The decoder has a simplified control flow designed using a bottom-up approach, that enables a number of low-level optimization that are typical of scientific applications, starting from highly optimized building blocks.

- *A second approach, based on the DotStar protocol processing tool, that is able to capture the essence of the OPRA structure in a handful of lines of a high-level description language*. This is combined with the same set of building blocks (actions) described above that, triggered by the protocol scanner.

- *Efficient parallel implementations of OPRA decoder that achieves processing rates more than an order of magnitude larger than the current needs of the market*. In the fastest configuration, our hand-optimized decoder achieves an impressive processing rate of 14.6 (3.4 per thread) millions messages/second with a single Intel Xeon E5472 quad-core socket. The DotStar protocol parser is only marginally slower than the hand-optimized parser, on average 7% on 5 distinct processor architectures. In the fastest single-socket configuration a single Intel Xeon E5472 quad-core is able to achieve a rate of 15 million messages/second.

- *An extensive performance evaluation* that exposes important properties of the DotStar OPRA protocol and parser, and analyzes the scalability of five reference systems, two variants of Intel Xeon, AMD Opteron, the IBM Power6, and the SUN UltraSPARC T2.

- *Insight onto the behavior of each architecture trying to explain where the time is spent by analyzing, for all the processor architectures under evaluation, each action associated to DotStar events*. All the action profiles are combined in a cycle-accurate performance model, that helps determine optimality of the approach, and to evaluate the impact of architectural or algorithmic changes for this type of workload.

- We design, analyze and optimize different high performance pseudo (such as Mersenne Twister) and quasi (such as Hammersley sequence) random number generators as well as normalization techniques on the Cell B.E, while maintaining high accuracy. Using these optimized kernels and Monte Carlo techniques we design efficient parallel algorithms for European Option pricing.

The rest of this chapter describes the OPRA feed format, the design of our decoder using DotStar protocol parser and an extensive body of performance evaluation. We also present our work on optimizing financial analytics later in the chapter.

4.1 Financial Feed Processing

4.1.1 Low Latency Trading

The growth in algorithmic and electronic trading has provided opportunities and exposed many challenges to financial institutions. The study of market data to predict market behaviour helps

these institutions increase profits. The performance of a trading process is critical in electronic trading. How fast a trading system can respond to the market will determine who wins and who loses. Every millisecond and every microsecond count in electronic trading. The first one to identify a trading opportunity generally doesn't leave much behind for the other players in the line. A few milliseconds difference in latency is enough to make difference between a profitable trade and a losing one, which can cause huge losses to the financial institutions and their clients. With such high stakes, the need for high speed and low latency trading system presents tremendous pressures to the technology division, in financial institutions, to optimize their market data messaging and trading system. Each stage of the ticker plant process adds unwanted latency, therefore high throughput and low latency processing at every stage is a critical factor to the success and competitive position of the market data processing system.

The latency in electronic trading is measured as the time it takes for a trade message to traverse the market data distribution network, starting from an exchange to when a trading application becomes aware of the trade. There are several sources that contribute to the latency in this messaging system. The first one is the physical distance between the trade application and the exchange. The market data cannot travel faster than the speed of light, thus financial institutions suffer increasing latency with increasing distance from the exchange. Switching network introduces another source of latency, latency due to store and forward, switch fabric processing, and frame queuing. The basic task of an Ethernet switch operation is to store the received data in memory until the entire frame is received. The switch then transmits the data frame to an appropriate output port. The latency of the store and forward operation depends on the frame size and the data rate. An Ethernet switch use queues in conjunction with the store and forward mechanism to eliminate the problem of frame collisions. When the load on a network is very light, latency due to queuing is minimal. OPRA data feeds are disseminated over 24 lines (48 when counting redundant delivery). Under a heavy network load, the switch will queue frames in buffer memory and introduce frame queuing latency. Typically, market data feeds terminate at ticker plants on the customer's premises. The ticker plants decode/process the market data stream, normalize it into a common format, and republish it for a wide variety of applications (e.g. trading application) through a messaging middleware. The delay introduced by the ticker plants and the messaging middleware also contributes to the latency in the

overall trading process.

With high speed and low latency a necessity for trading processes in financial market, there is continuing pressure for investments by financial institutions in low-latency trading and market data architectures. To increase profits, new applications are being deployed on these low latency architectures. To fight with the latency attributed to the physical distance, companies such as Activ Financial and TelX are offering low latency collocation services to financial institutions. It also saves the financial institutions from building/expanding their own facilities to house the network architecture. Direct exchange feed solutions that connect from customer's site directly to the exchange are another way to reduce the latency. As opposed to a data consolidator, the direct-exchange feed eliminates data hops and thus reduces the latency. Re-evaluation of ticker plant and messaging middleware is another place to look for latency reduction. Many financial institutions are still using in-house ticker plants to handle the market data. As market data rates continue to increase, we are not just talking about low latency, but low latency at high-throughput rate. It also comes with the issues of system scalability, power consumption, and consolidation. Managing in-house ticker plant becomes challenging for the market data technology staff. The traditional software-based ticker plants on commodity white boxes seem to be breaking. There are vendors doing this in hardware to accelerate the ticker plant functionality. For example, FPGA is used to build an acceleration appliance in a reconfigurable way for ticker plant functionality.

4.1.2 OPRA feed decoding

An essential component in ensuring the timely reporting of option equity/index and every other transaction is the OPRA IP multicast data stream. OPRA messages are delivered through the national market system with a UDP-based IP multicast [129]. The options market data generated by each participant is assembled in prescribed message formats and transmitted to the appropriate TCP/IP processor address via participant's private communications facility. As each message is received, it is merged with messages received from other participants, and the consolidated message stream is transmitted simultaneously to all data recipients via their private communications facilities. Each message is duplicated and delivered to two multicast groups. OPRA messages are divided into 24 data lines (48 when counting redundant delivery) based on their underlying symbol. Multiple

OPRA messages are encapsulated in a block and then inserted in an Ethernet frame. The original definition of OPRA messages is based on an ASCII format [127], which uses only string based encoding and contains redundant information. With the growth of data volume, a more compact representation for messages was introduced: OPRA FAST (FIX Adapted for STreaming) [126, 128].

The techniques used in the FAST protocol include implicit tagging, field encoding, stop bit, and binary encoding (see Tables 5 and 6). Implicit tagging eliminates the overhead of field tags transmission. The order of fields in the FAST message is fixed and thus the meaning of each field can be determined by its position in the message. The implicit tagging is usually done through XML-based FAST template. The presence map (PMAP) is a bit pattern at the beginning of each message where each bit is used to indicate whether its corresponding field is present. Field encoding defines each field with a specific action, which is specified in a template file. The final value for a field is the outcome of the action taken for the field. Actions such as "copy code", "increment", and "delta" allow FAST to remove redundancy from the messages. A stop bit is used for variable-length coding, by using the most significant bit in each byte as a delimiter. FAST uses binary representation, instead of text string, to represent field values. OPRA adopted FAST protocol for reducing the bandwidth needed for OPRA messages.

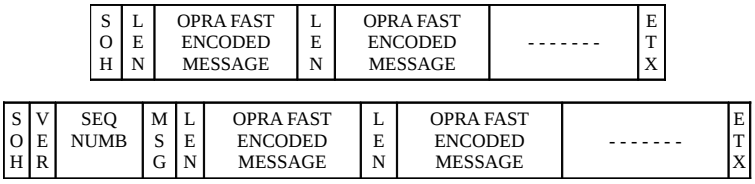

Figure 35: OPRA FAST encoded packet format: (a) Version 1.03 (b) Version 2.0

Figure 35(ab) shows the format of an encoded OPRA version 1 packet. Start of Header (SOH) and End of Text (ETX) are two control characters that mark the start and end of the packet. One transmission block can contain multiple messages, where a message is a unit of data that can be independently processed by the receiver. In OPRA FAST version 2 (see Figure 35(b)) there is a header after SOH and before the first message to further reduce redundant information. The first byte of an encoded message contains the length in bytes and is followed by the presence map. For example, presence map 01001010 means field 1, field 4 and field 6 are present. The type of each field is specified by message category and type. Data fields that are not present in an encoded

message but that are required by the category will have their value copied from the same field of a previous messages and optionally incremented. OPRA data fields can be either unsigned integer or string.

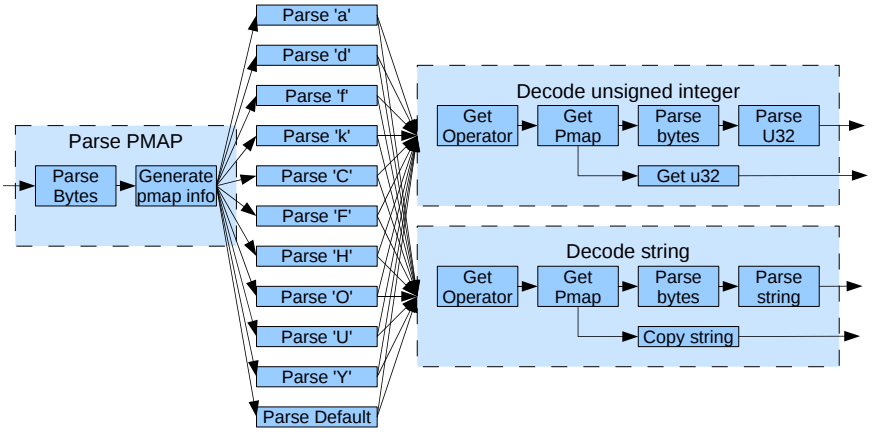

Figure 36: OPRA Reference decoder block diagram.

Figure 36 shows a block diagram of the reference OPRA FAST decoder provided along with the standard. The implementation starts by creating a new message and parsing the presence map, computing its length by checking the stop bit of every byte until one is found set, masking the stop bit and copying all the data into temporary storage. The presence map bits are then examined to determine which fields are present and which require dictionary values. The implementation proceeds checking the category of the message and calls a specific decoder function, where the actual decoding for each required field is implemented. There are two basic building blocks: decoding unsigned integers and decoding strings, both examine PMAP information, input data and manipulate the last value dictionary. We initially tried to optimize the reference decoder using a *top-down* approach, by speeding up the functions that are taking more time-consuming. Unfortunately the computational load is distributed across a large number of functions, as shown in Figure 37, and our effort resulted in a very limited performance improvement.

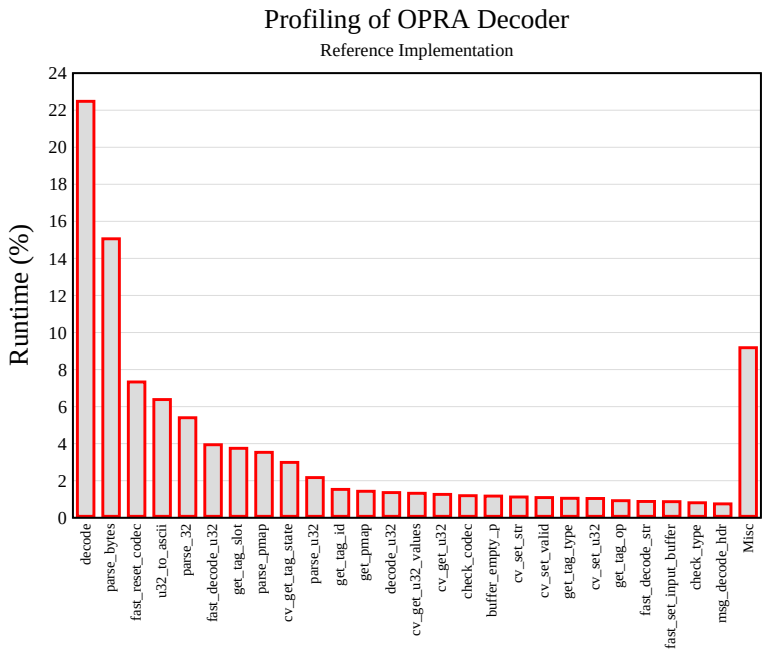

Figure 37: Profiling of the reference OPRA decoder

4.1.3 A Streamlined Bottom-Up Implementation

We then designed another version of the OPRA decoder following a bottom-up approach, trying eliminate the complex control flow structure existing in the reference implementation. In this approach we identify important computationally intensive kernels, optimize these routines independently and analyze the assembly-level code to get best performance. These optimized routines are then crafted together to perform OPRA feed decoding as shown in Figure 39.

To further improve the performance of these kernels, we introduce important novel optimizations to the OPRA FAST decoding algorithm, which we describe below. First, we replace branches and category-specific routines, by a single routine that uses a category-specific bitmap. A category bitmap is a bit stream signifying all the required bits of that category, where the bit is on whenever the corresponding field is needed by the category. Once the category of a message has been determined, this bitmap is passed to the next building block for specifying the fields that are relevant to a message of this category.

The second design choice is related to presence map parsing. The reference code parses the PMAP field bit by bit, testing every presence bit to see if it is set or not. However, since only a

Table 5: OPRA message categories with description

	Description
Category 'a'	Equity and Index Last Sale
Category 'd'	Open Interest
Category 'f'	Equity and Index End of Day Summary
Category 'k'	Index and Stock Quotes
Category 'C'	Administrative
Category 'F'	FCO End of Day Summary
Category 'H'	Control
Category 'O'	FCO Last Sale
Category 'U'	FCO Quote
Category 'Y'	Underlying Value Message
default	Contains Text value

subset of fields are required for each message category, it is tedious to check for each corresponding bit. Since PMAP points to the location of all the present fields, it can also be considered as a field map for the data fields that needs to be updated. To get the field map for the rest of data fields that are not present but required, we can simply create another bitmap, through an *xor* of PMAP and category-specific bitmap. This bitmap gives information of all the fields that need to be copied from last known values. After computing these bitmaps, we use *clz* (count leading zeroes) operation to determine the field ID of the next field present in the message block. This operation can also be used on the copy bitmap to determine the field IDs of the fields that need to be copied. Figure 38 provides an overview of the overall decoding algorithm.

The third optimization deals with the string field processing. If a string field relevant to a category is not in the message block, it is copied from last known value. Instead of copying the string value multiple times, we create a separate buffer, copy the string once to this buffer and reference that to the decoded messages through an array offset. As shown in Figure 39, the bottom up algorithm has a much simpler control flow structure than the reference decoder described in Figures 36(a) and 36(b).

4.1.4 High-Level Protocol Processing with DotStar

A fundamental step in the semantic analysis of network data is to parse the input stream using a high-level formalism. This process transforms raw bytes into structured, typed, and semantically meaningful data fields that provide a high-level representation of the traffic. Constructing a protocol parsers by hand is a tedious process and it is error-prone due to the complexity of the low level

Table 6: Fields in OPRA message

Field Name	ID	Encode Operator	Data Type	Categories									
				a	d	f	k	C	F	H	O	U	Y
MESSAGE_CATEGORY	0	COPY CODE	Unsigned Integer	x	x	x	x	x	x	x	x	x	x
MESSAGE_TYPE	1	COPY CODE	Unsigned Integer	x	x	x	x	x	x	x	x	x	x
PARTICIPANT_ID	2	COPY CODE	Unsigned Integer	x	x	x	x	x	x	x	x	x	x
RETRANSMISSION_REQUESTER	3	COPY CODE	Unsigned Integer	x	x	x	x	x	x	x	x	x	x
MESSAGE_SEQUENCE_NUMBER	4	INCREMENT	Unsigned Integer	x	x	x	x	x	x	x	x	x	x
TIME	5	COPY CODE	Unsigned Integer	x	x	x	x	x	x	x	x	x	x
SECURITY_SYMBOL	6	COPY CODE	STRING	x	x	x	x		x		x	x	
EXPIRATION_MONTH	7	COPY CODE	Unsigned Integer	x	x	x	x						
EXPIRATION_DATE	8	COPY CODE	Unsigned Integer	x	x	x	x		x		x	x	
YEAR	9	COPY CODE	Unsigned Integer	x	x	x	x						
STRIKE_PRICE_DENOMINATOR_CODE	10	COPY CODE	Unsigned Integer	x	x	x	x						
EXPLICIT_STRIKE_PRICE	11	COPY CODE	Unsigned Integer	x	x	x	x						
STRIKE_PRICE_CODE	12	COPY CODE	Unsigned Integer	x	x	x	x		x		x	x	
VOLUME	13	COPY CODE	Unsigned Integer	x		x			x		x		
OPEN_INT_VOLUME	14	COPY CODE	Unsigned Integer		x	x			x				
PREMIUM_PRICE_DENOMINATOR_CODE	15	COPY CODE	Unsigned Integer	x		x	x						
PREMIUM_PRICE	16	COPY CODE	Unsigned Integer	x							x		
OPEN_PRICE	17	COPY CODE	Unsigned Integer			x			x				
HIGH_PRICE	18	COPY CODE	Unsigned Integer			x			x				
LOW_PRICE	19	COPY CODE	Unsigned Integer			x			x				
LAST_PRICE	20	COPY CODE	Unsigned Integer			x			x				
NET_CHANGE_INDICATOR	21	COPY CODE	Unsigned Integer			x			x				
NET_CHANGE	22	COPY CODE	Unsigned Integer			x			x				
UNDERLYING_PRICE_DENOM	23	COPY CODE	Unsigned Integer			x							
UNDERLYING_STOCK_PRICE	24	COPY CODE	Unsigned Integer			x			x				
BID_PRICE	25	COPY CODE	Unsigned Integer			x	x				x		
BID_SIZE	26	COPY CODE	Unsigned Integer				x						
OFFER_PRICE	27	COPY CODE	Unsigned Integer			x	x				x		
OFFER_SIZE	28	COPY CODE	Unsigned Integer				x						
SESSION_INDICATOR	29	COPY CODE	Unsigned Integer	x			x						
BBO_INDICATOR	30	COPY CODE	Unsigned Integer				x						
BEST_BID_PARTICIPANT_ID	31	COPY CODE	Unsigned Integer				x						
BEST_BID_PRICE_DENOMINATOR_CODE	32	COPY CODE	Unsigned Integer				x						
BEST_BID_PRICE	33	COPY CODE	Unsigned Integer				x						
BEST_BID_SIZE	34	COPY CODE	Unsigned Integer				x						
BEST_OFFER_PARTICIPANT_ID	35	COPY CODE	Unsigned Integer				x						
BEST_OFFER_PRICE_DENOMINATOR_CODE	36	COPY CODE	Unsigned Integer				x						
BEST_OFFER_PRICE	37	COPY CODE	Unsigned Integer				x						
BEST_OFFER_SIZE	38	COPY CODE	Unsigned Integer				x						
NUMBER_OF_INDICES_IN_GROUP	39	COPY CODE	Unsigned Integer										x
NUMBER_OF_FOREIGN_CURRENCY_SPOT_VALUES_IN_GROUP	40	COPY CODE	Unsigned Integer										x
INDEX_SYMBOL	41	COPY CODE	STRING										x
INDEX_VALUE	42	COPY CODE	STRING										x
BID_INDEX_VALUE	43	COPY CODE	STRING										x
OFFER_INDEX_VALUE	44	COPY CODE	STRING										x
FCO_SYMBOL	45	COPY CODE	STRING										x
DECIMAL_PLACEMENT_INDICATOR	46	COPY CODE	Unsigned Integer										x
FOREIGN_CURRENCY_SPOT_VALUE	47	COPY CODE	Unsigned Integer										x
TEXT	48	COPY CODE	STRING					x		x			
DEF_MSG	49	COPY CODE	STRING										

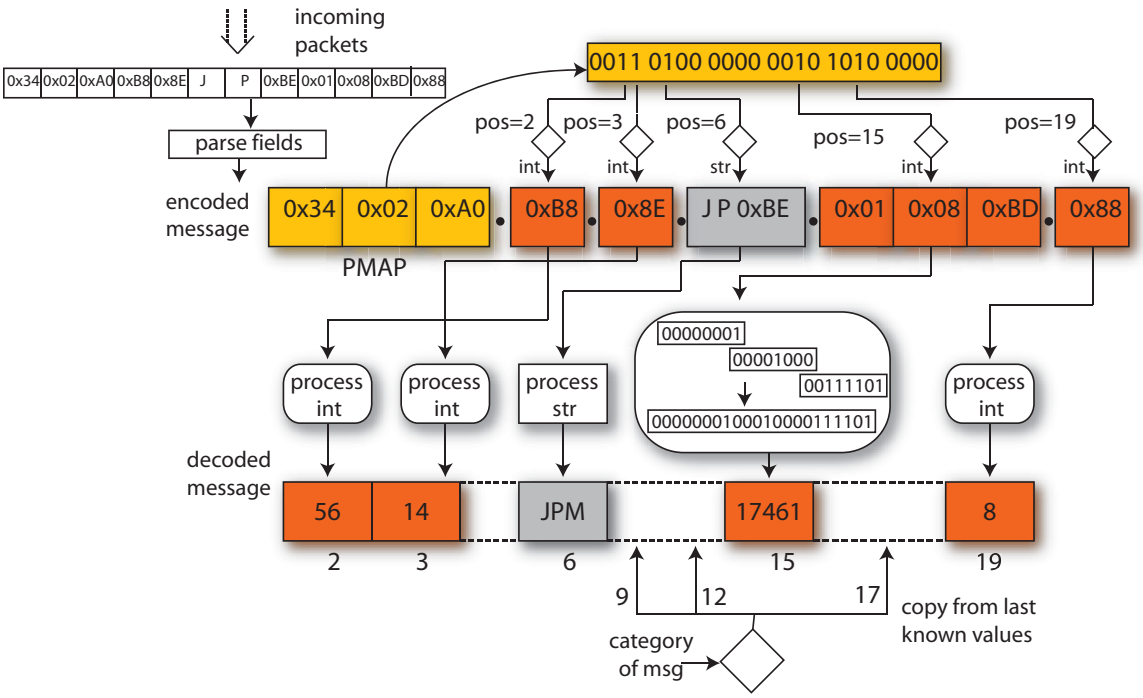

Figure 38: Presence and field map bit manipulation.

details in most protocols. Moreover optimizing the performance of the implementation for one or more target architecture is very complex and can be challenging; finally any change to the protocol, like a new version or new fields, may require extensive rewrite of the parser code.

A large amount of work approached this issues by using declarative languages to describe data layout and transfer protocols. Examples are Interface Definition Languages (IDL), like XDR [134], RPC [133] or ASN.1 [88], and regular languages and grammars, like BNF, ABNF [108], lex and yacc, or declarative languages like binpac [109]. These solutions are often tailored for a specific class of protocols, like binary data or text oriented communication. A common characteristic of all these solutions is that they are very high level and "elegant", and provide the user with great expressiveness, but the generated parser performance is often one order of magnitude slower than an equivalent handcrafted one, and may be unable to parse real-time heavy volume applications.

While exploring how to extend our fast keyword scanner automaton in order to handle regular expression sets, we developed DotStar [110], a complete tool-chain that builds a deterministic finite automaton for recognizing the language of a regular expression set. The generated automaton is an extension of the Aho-Corasick [102], the de facto standard for fast keyword scanning algorithms.

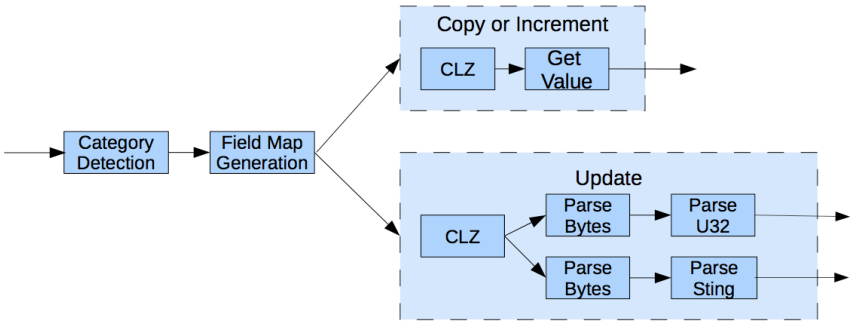

Figure 39: Bottom-up reference decoder block diagram.

The Aho-Corasick algorithm operates on a keyword tree that contains a combined representation of all keywords. The keyword tree is transformed into a Deterministic Finite Automaton (DFA) by creating a failure function $F()$ that is followed when no direct transition is available; $F()$ points to the longest proper suffix already recognized.

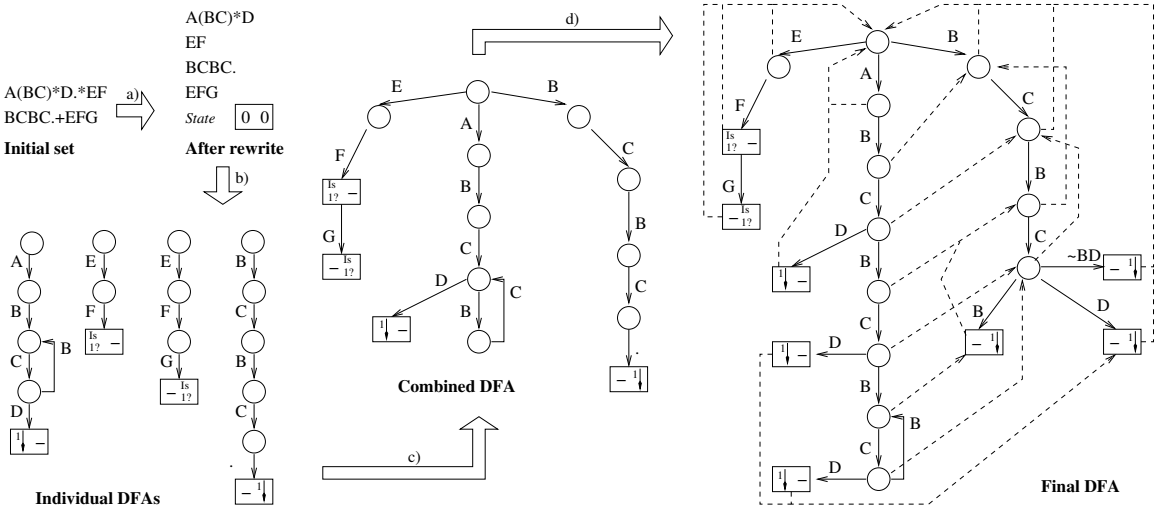

Figure 40: A graphical representation of DotStar compiler steps: a) is the pre-processor, which reduces the number of states, step b) builds individual automata for each regular expression, c) combines them in a unique automaton and d) computes the failure function.

Finding every instance of a regular expression pattern, including overlaps, is a "complex" problem either in space or time [93]. Matching a complete set of regular expressions adds another level of complexity: DotStar employs a novel mechanism for combining several regular expressions into a single engine while keeping the complexity of the problem under control. DotStar is based on a sophisticated compile time analysis of the input set of regular expression and on a large number of

automatic transformations and optimizing algorithms. The compilation process proceeds through several stages (see Figure 40): at first each regular expression is simplified in a normal form, rewriting it and splitting it into sub-expressions and it is transformed into a Glushkov [61] NFA. We selected the Glushkov representation because it has a number of interesting properties [34]. The Glushkov NFA is then turned into a DFA, These automata are then combined together by an algorithm that operates on their topology. The resulting graph is then extended, as in the Aho-Corasick algorithm, by a "failure" function, whose computation can further modify the graph structure. The result is a single pass deterministic automaton that:

- groups every regular expression in a single automaton,
- reports exhaustive and complete matches, including every overlapping pattern,
- in most practical cases it is as memory efficient as an NFA

During the compilation process DotStar can be tuned and optimized for a specific architecture: for example it is possible to trade the number of states with the size of the state bit word, or it is possible to reduce the size of the automaton using parallel counters if the target architecture support vector instructions.

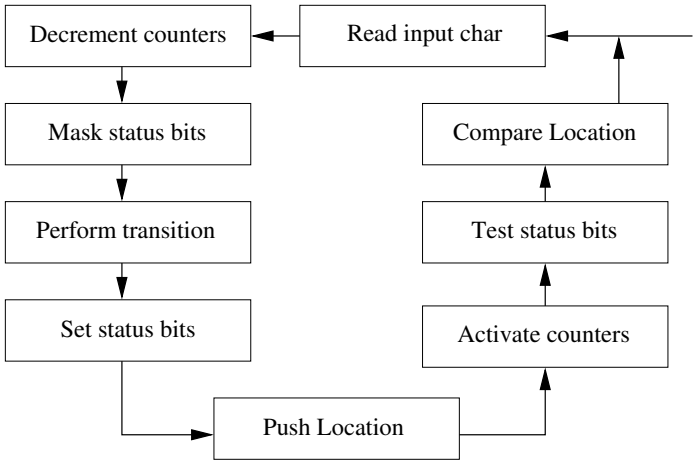

Figure 41: Various steps in DotStar runtime

The DotStar runtime shares the same execution model of the well known Aho-Corasick algorithm and : for each symbol the automaton transitions at most once across a keyword tree edge or a finite number of times across F() edges until either a proper suffix is detected or the root node is

reached. A linear time DFA is obtained by pre-computing every F(). DotStar extends this runtime by adding actions to be executed when a state is entered. These actions have been defined in such a way as to be parametric and exploit modern CPU features, thus allowing extensive runtime optimizations targeted at a specific architecture class. Our runtime system, depicted in Figure 41, leverages the available memory hierarchy, pipeline and multiple cores using caching, partitioning, data replication and mapping [147, 122, 145]; it also employs automata interleaving, bit-level parallelism and vector instructions to operate on data structures.

DotStar solution is applicable in several fields, from network intrusion detection to data analysis and also as a front end for a protocol parser, since the first logical step for any protocol parser is recognizing where data fields start and end. For supporting this task we defined a simple declarative language, called DSParser, that describes a data layout as a sequence of regular expression fragments "connected" using standard imperative constructs, like if/then/else/while. Actions can be inserted after a (partial) expression is recognized; these actions are either *system actions*, that perform common operations or user defined functions on blocks of input data. A precompiler tool parses the declarative language and builds a suitable regular expression set that is compiled used the DotStar toolchain. A small number of states of the resulting automaton is then annotated with the system and user defined actions specified in the initial protocol definition. The DSParser source code for analyzing OPRA V2 messages is remarkably simple, compact and easy to manage:[1]

```
MATCH "......................................."
MATCH "\x01\x02"
PUSH
MATCH ".........."
EXECUTE sequence_number
MATCH "..."
LOOP
  WHILECNT "."
    PUSH
    MATCH "[\x00-\x7f]*[\x80-\xff]"
    EXECUTE action_pmap
    PUSH
    WSWITCH
      CASE "[\x80-\xff]"
```

[1] It is worth noting that the full OPRA FAST protocol is described in a 105-pages manual!!

```
        EXECUTE action_field
PUSH
ENDCASE
      CASE "[\x00-\x7f]"
        MATCH "[\x00-\x7f]*[\x80-\xff]"
        SEND 0x02 0
        EXECUTE action_field
        ENDCASE
    ENDWSWITCH
  ENDWHILECNT
ENDLOOP
```

Intuitively the resulting automaton will start by skipping the IP/UDP headers and will match OPRA v2 start byte and version number. It will then mark the stream position for the successive action and recognize the initial sequence number for the packet, calling the appropriate user defined code; after that it will loop examining every message in the packet and detecting first the PMAP and then all individual fields. User defined actions operate on blocks of input data recognized by the parser automaton and the overall system processing model proceeds along the following high level steps:

1. look for an interesting section of data: the automaton reads input symbol(s) and switches state until a PUSH action is reached;
2. save the start of interesting data: the current stream position is saved in state machine memory;
3. look for the end of a data field: the automaton reads more input symbols until a state with a user defined action is found;
4. handle the data field: the user defined action is invoked over the block of data from the saved position to the current stream position.

Thanks to DotStar's arsenal of optimization and configuration parameters, we can fine-tune the compiled automaton and we can select how this automaton "connects" with user defined actions, for example our modular runtime can either invoke user actions when detected, for minimum latency, or can "schedule" their execution at a later moment, eventually by a different thread for maximum

throughput (and this will allow automata interleaving for better CPU pipeline utilization). The individual user actions are exactly the basic building blocks used in the hand made parser for converting integers and PMAP and parsing PMAP content, as shown in Figure 38. The rest of the decoder, which takes care of input handling, data sequencing and field delimitation, its handled by DSParser compiler and is automatically tuned for the architecture, exploiting optimization that are difficult to implement in the hand-optimized code.

4.1.5 Experimental Results

OPRA market data feeds are transmitted in a set of 24 channels. In this experimental section, we assume that each channel can inject messages at full speed by storing the OPRA feeds in main memory, and therefore is not the bottleneck of the decoder. While this hypothesis may not be realistic in practice, it serves the purpose of pushing to the limit the OPRA decoding algorithm and provides an upper bound.

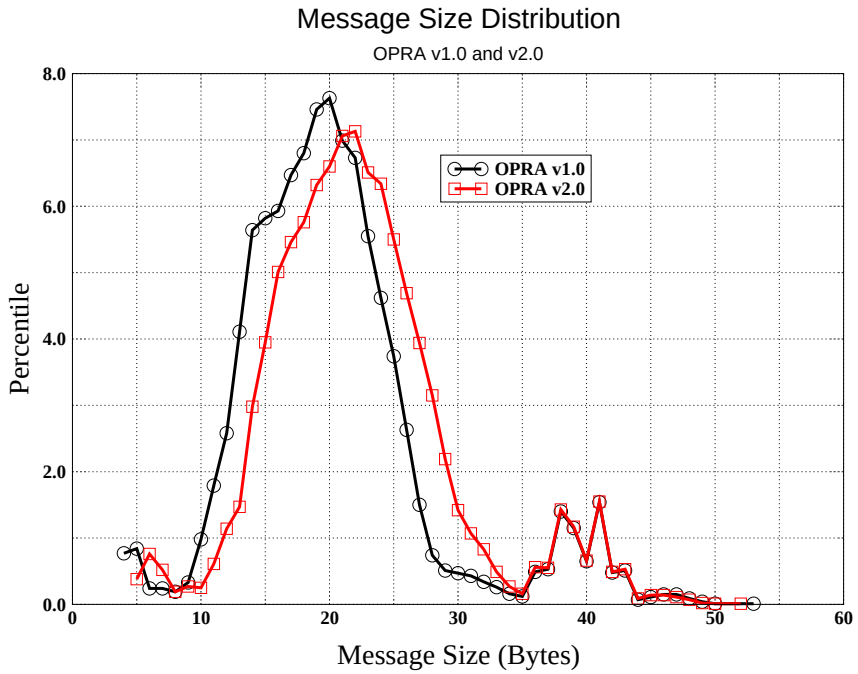

Figure 42: OPRA Message Size Distribution

OPRA packets are 400 bytes on average, each containing multiple messages that are encoded using the FAST protocol. We use real market data feeds, obtained by capturing few seconds of the network traffic, totalling one gigabyte of version 1 and 2 OPRA format for experimental analysis

throughout this section.

Figure 42 gives the distribution of the message size for both OPRA version 1 and 2 format. The messages are typically distributed across the 10-50 byte range, with an average message size of 21 bytes. Thus each packet contains 19 messages on average.

Under the assumption of full injection, the feeds across the various channels have very similar data pattern and distribution and, as shown in Figure 43, they tend to have the same processing rate. Since, the performance is insensitive to the OPRA protocol version, we will consider only OPRA version 2 traces in the rest of this chapter.

Figure 44 gives a distribution of the OPRA messages among 11 categories, as described earlier. We observe that 99% messages flowing in the market are category K equity and index quotes. The first field of each message after the *message length* is the PMAP, containing encoded information about the position and type of the data fields that follow. Figure 45 shows a distribution of the PMAP length, for version 2 OPRA format, and shows that a majority of the messages contain a 5 byte PMAP. The data fields can be either integer or string type, that is given as a part of the protocol specification. Each OPRA message can contain multiple integer fields, with length varying from 1 to 5 bytes. Figure 46 gives a distribution of the encoded integer field length, and shows that more than 80% of encoded integers are less than 2 bytes.

In this chapter, we present an extensive performance analysis of our algorithm on a variety of multi-core architectures. In our tests, we use Intel Xeon Q6600 (Quad), Intel Xeon 5472 (Quad, a.k.a. Harpertown), AMD Opteron 2352 (Quad), IBM Power 6, and Sun UltraSparc T2 (a.k.a. Niagara-2). Power6 and Niagara-2 are hardware multithreaded with 2 threads and 8 threads per core, respectively. Table 7 gives more information about the CPU speed, number of cores, threads, cache size, sockets on each architecture. Here, we discuss the performance of the three approaches for processing OPRA FAST messages, the top-down reference implementation, the hand-optimized bottom-up version and DotStar as discussed in sections 4.1.2, 4.1.3 and 4.1.4, respectively. Figure 47 gives the decoding rate per processing thread in millions of messages/second using the three approaches on the multi-core architectures described in Table 7. We observe that our bottom-up and DotStar implementations are consistently 3 to 4 times faster than the reference implementation; the Intel Xeon E5472 gives the maximum processing rate for a given thread, and that our

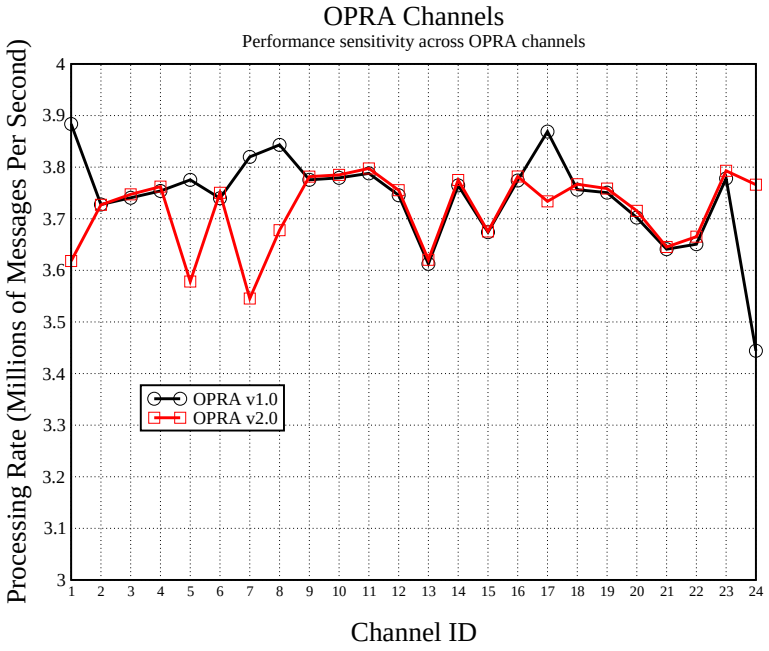

Figure 43: Channel Sensitivity: the processing rate is almost insensitive to the Channel ID and OPRA version.

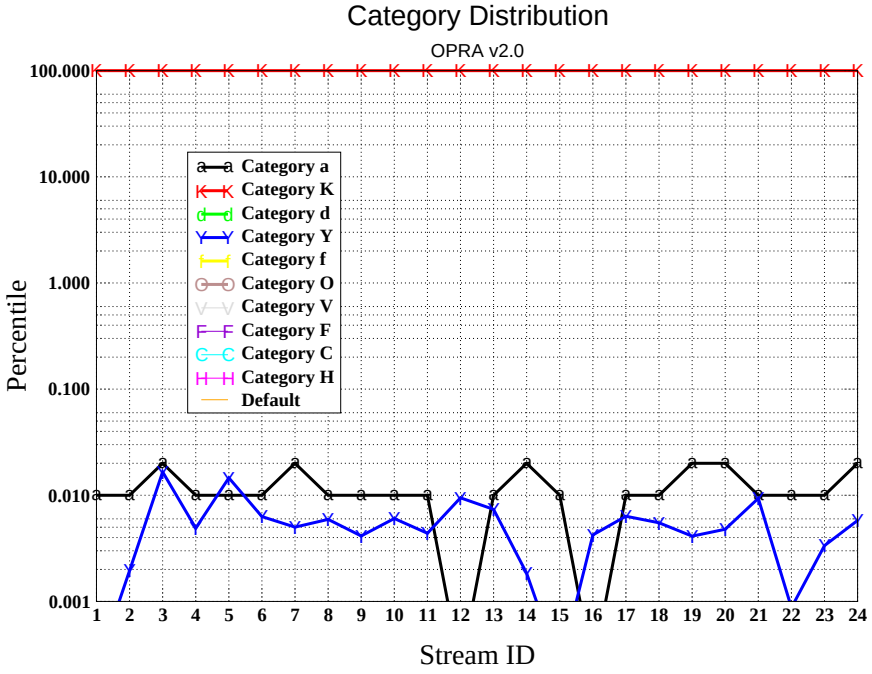

Figure 44: OPRA message category distribution

optimized Bottom-Up and DotStar implementations are similar in terms of performance, with the hand-optimized version that is only 7% faster, on average. The Sun UltraSparc T2 processor is designed to operate with multiple threads per core, thus the single thread performance is much slower than other processors. It is also important to note that our single core OPRA decoding rate is much higher than the current needs of the market, as given in Figure 34.

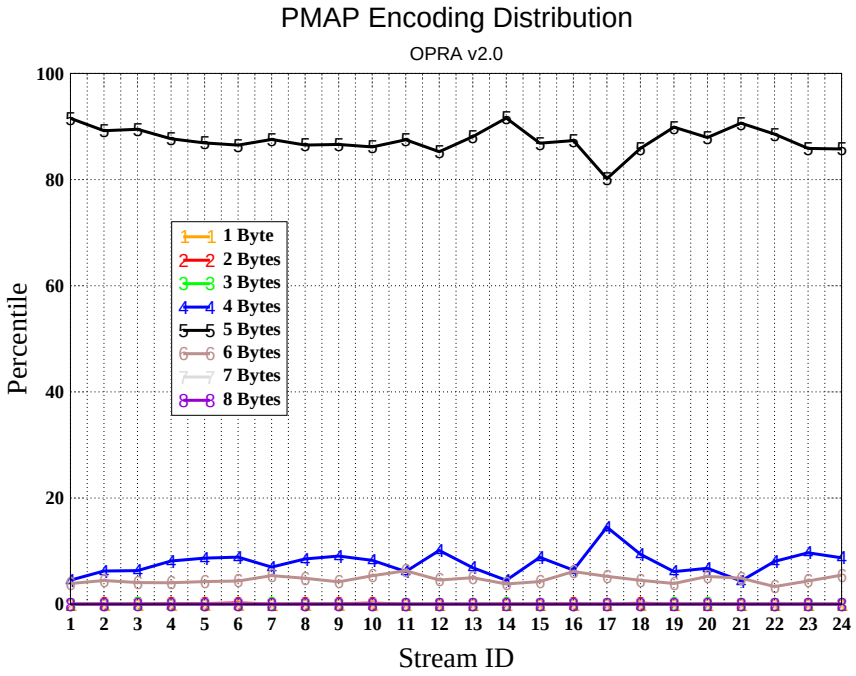

Figure 45: Encoded OPRA pmap field length distribution

Figure 48 presents a scalability study of our high-level DotStar implementation across threads/-cores on the five multi-core architectures. For the Intel Xeon processors our implementation scales almost linearly and processes on a single socket, 15 million messages/second, on the E5472 and 12 million messages/second, on the Q6600. On Niagara-2, the performance scales linearly up to 16 threads, and reaches 6.8 million messages/second using 64 threads (8 threads/8 cores). We believe we start hitting the memory bandwidth wall beyond 32 threads. For the IBM Power6, our single thread performance is not as impressive as the Intel E5472, but gets a scaling advantage upto 16 threads using 8 cores, giving a performance of 26 million messages/second. The performance scales linearly up to 8 threads (1 thread per core), with a performance benefit of 1.5x up to 16 threads.

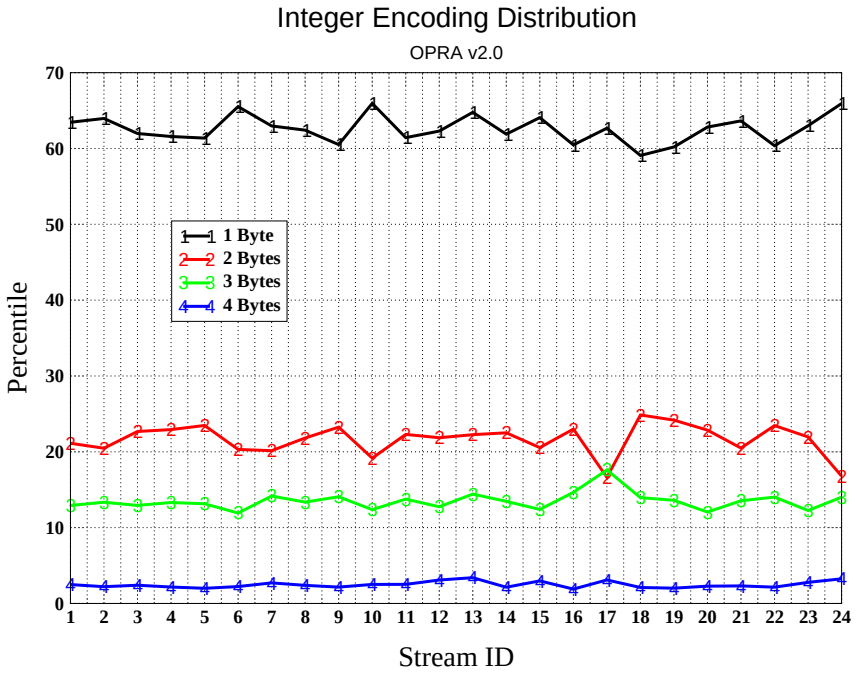

Figure 46: Encoded OPRA integer field length distribution

Table 7: OPRA performance analysis: Architecture configurations

	CPU Speed (GHz)	Sockets	Cores per Socket	Threads per Core	Threads	Cache Size (KB)
INTEL Xeon Q6600	2.4	1	4	1	4	4096
INTEL Xeon E5472	3.0	1	4	1	4	6144
AMD Opteron 2352	2.1	1	4	1	4	512
Sun UltraSPARC T2	1.2	1	8	8	64	4096
IBM Power6	4.7	4	2	2	16	4096

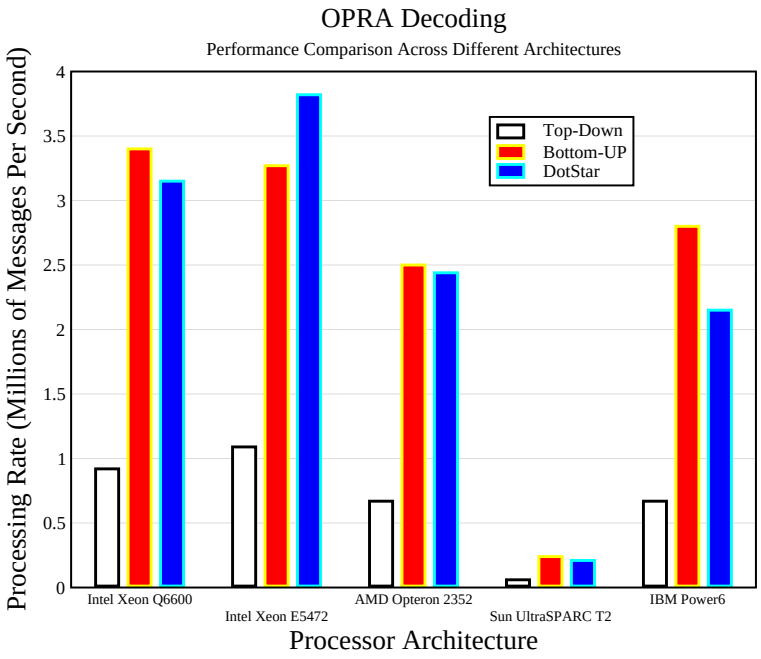

Figure 47: Performance Comparison of the various OPRA decoding approaches

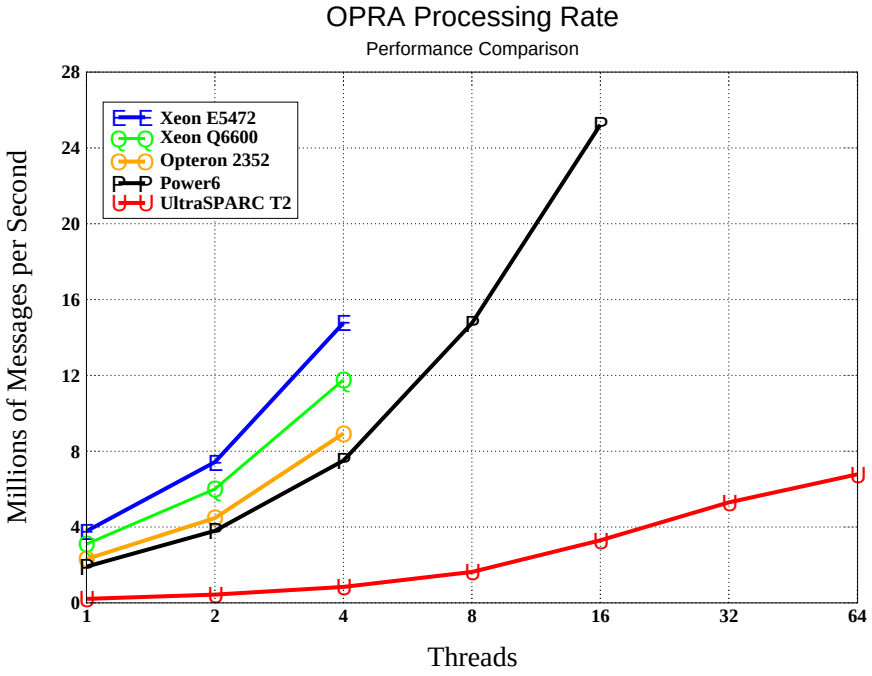

Figure 48: Scalability of our OPRA decoder on various architectures

Table 8: Breakdown of runtime : Optimality Analysis

			INTEL Xeon Q6600			INTEL Xeon E5472			AMD Opteron 2352			SUN UltraSPARC T2			IBM Power6		
	size	%	instr	ns	cyc	instr	ns	cyc	instr	ns	cyc	instr	ns	cyc	instr	ns	cyc
INT	1	62.6	3.0	1.4	3.4	3.0	1.5	4.5	3.0	4.0	8.4	3.0	6.5	7.8	4	2.3	10.8
	2	22.1	3.5	1.3	3.2	3.5	1.6	4.9	3.5	4.8	10.1	3.0	6.8	8.2	3.5	2.2	10.3
	3	13.6	3.7	1.4	3.4	3.7	1.7	5.2	3.7	5.0	10.5	3.0	5.4	6.5	3.3	2.0	9.4
	4	2.7	3.7	1.5	3.6	3.7	1.8	5.4	3.7	5.1	10.7	3.0	6.6	7.9	3.2	1.9	8.9
	5	0.0	3.8	1.5	3.8	3.8	1.8	5.5	3.8	5.1	10.8	3.0	6.6	7.9	3.2	1.9	8.8
PMAP	1	0.0	7.0	2.6	6.3	7.0	3.3	9.9	7.0	9.4	19.7	8.0	17.3	20.7	7	4.4	20.7
	2	0.0	6.0	2.5	6.0	6.0	3.0	9.0	6.0	8.0	16.9	7.4	16.5	19.8	6.5	4.2	19.7
	3	0.0	5.7	2.5	6.0	5.7	2.8	8.5	5.7	7.6	15.9	7.3	16.0	19.2	6.3	4.1	19.0
	4	8.1	5.5	2.4	5.9	5.5	2.8	8.3	5.5	7.4	15.5	7.5	16.9	20.3	6.2	3.9	18.6
	5	87.1	5.4	2.4	5.8	5.4	2.7	8.1	5.4	7.3	15.4	7.2	15.9	19.1	6.2	3.9	18.1
	6	4.8	5.3	2.3	5.6	5.3	2.7	8.0	5.3	7.1	15.0	7.7	17.5	21.0	6.0	3.8	17.8
	7	0.0	5.2	2.3	5.5	5.2	2.6	7.8	5.2	7.0	14.7	7.9	18.2	21.8	6	3.8	17.9
	8	0.0	5.1	2.3	5.5	5.1	2.6	7.7	5.1	6.9	14.4	7.5	16.9	20.3	6	3.9	18.1
Dcopy/msg			-	11.0	26.4	-	15.0	45.0	-	45.1	94.7	-	706.0	847.2	-	14.0	65.8
Scopy/byte			-	1.4	3.4	-	1.9	5.7	-	4.5	9.5	-	15.2	18.2	-	3.9	18.6
CLZ/field			6	2.8	6.7	5	2.0	6	5	4.1	8.5	17	36.9	44.3	6	4	18.8
			msgs M/s	ns /b	cyc /b	msgs M/s	ns /b	cyc /b	msgs M/s	ns /b	cyc /b	msgs M/s	ns /b	cyc /b	msgs M/s	ns /b	cyc /b
Basic blocks peak			11.8	3.6	8.5	11.8	3.5	10.6	4.7	9.0	19.0	0.7	59.0	70.8	7.5	5.6	26.2
Actions only			8.8	4.8	11.5	9.5	4.4	20.6	4.3	9.7	20.3	0.6	69.3	83.2	5.3	7.9	37.2
DotStar rate only			5.8	8.1	19.4	7.5	6.4	30.0	5.7	7.5	15.7	0.4	109.8	131.8	4.1	11.2	52.6
Optimal			3.5	-	-	4.2	-	-	2.5	-	-	0.24	-	-	2.3	-	-
Actions w/DotStar			**3.1**	13.3	32.0	**3.8**	11.0	51.7	**2.4**	17.5	36.8	**0.2**	200.0	240.0	2.1	20.0	94.0
Optimality ratio			**0.89**	-	-	**0.91**	-	-	**0.96**	-	-	**0.84**	-	-	**0.92**	-	-

4.1.6 Discussion

In this section we provide insight into the performance results, and present a reference table (Table 8) that explains in detail where the time is spent. The OPRA FAST decoding algorithm can be broken down into 5 main components, processing of the PMAP, *integer*, *string* fields, *copying previous values*, and *computing field index from PMAP*. Every OPRA message contains one PMAP field with an average size of 5 bytes, *integer* fields of 13 bytes, *string* fields of 4 bytes, in total 22 bytes.

Decoding a message requires *copying* the last known values into a new data structure, as the encoded message only contains information on a subset of fields. To find the index of each new data fields, we perform a *clz* (count leading zeroes) operation. For each of these actions we calculate the instructions from assembly, Table 8 give the instructions/byte for processing the PMAP, *integer* and *string* fields, on each multicore processor. For nsec and cycles per byte we compute actual performance, by performing multiple iterations of the algorithm with and without the corresponding action and normalizing the difference. Similarly, for the *clz* action, we compute cycles, nsec, and instructions per data field, and for the *copy msg* we compute average nsec and cycles per message.

Using the field length distributions, their occurrence probability rates, and individual action performance, we compute the aggregate estimated peak performance in the first row of the last block in the table. Note that this peak estimate does not take into account any control flow or I/O time. In the second row we give the actual actions-only performance that is computed by taking the different between the total performance and performance after commenting out the actions, lets call it A. The DotStar rate is the peak performance of our parsing algorithm on OPRA feeds, that does not include any actions on the recognized data fields, lets call it B.

The *optimal* row in that block is the estimated peak performance that we should have got by combining the DotStar routine and our optimized-actions routines, that is computed using the formula $(\frac{1}{A} + \frac{1}{B})^{-1}$. We compare this to our actual performance and compute the *optimality ratio* in the last row of the table.

We notice that:

- On the IBM Power 6, we get similar nsec/byte performance as compared to Intel Xeon processor. We believe that our code is not able to take advantage of the deep pipeline and in-order

execution model of this CPU.

- To get maximum performance for the *copy msg* routine, we developed our own vectorized memory copy using SSE intrinsics, and pipelined load and stores to obtain best performance. On the Sun Niagara-2 we used a manual copy, at the granularity of an integer, and unrolled the load and store instructions. With vectorized memory copy we get about 2x performance advantage over the *memcpy* library routine, whereas for the manual scalar copy we get an advantage of 1.2x on Sun Niagara-2.

- On Sun-Niagara, the *copy* action is much slower than on other processors, that leads to a significant performance penalty.

- Table 9 gives the latency for processing/decoding an OPRA message on each of the multicore processors discussed in this chapter. We decode one message at a time on a single processing thread, thus the average latency to process a message is given by taking the inverse of the processing rate. We observe a latency between 200 and 500 nsec on the Intel/IBM/AMD processors, which accounts for only a negligible latency inside the ticker plant, under a very high throughput.

- Our implementation requires integer/string processing as opposed to floating point computation, this fails to utilize the much improved floating point units on the Sun Niagara-2 processor.

- On Intel Q6600 processor, the cycles/byte performance matches closely with the instructions/byte count.

- Our implementation is close to the optimal performance we could obtain with the high-level approach, with an average optimality ratio of 0.9.

4.2 Financial Data Analytics

High performance computing is critical in the Financial Sector to aid complex analytics in financial models. These models aim to understand and deal with uncertainties prevalent in the market. The use of technologies such as multicore processors, clusters and grid computing is well established in

Table 9: OPRA message processing latency using our algorithm

	Latency/msg (nsec)
INTEL Xeon Q6600	317
INTEL Xeon E5472	261
AMD Opteron 2352	409
Sun UltraSPARC T2	4545
IBM Power6	476

this market, with growing interest in the Cell/B.E., GPUs and FPGAs [67]. The IBM Cell/B.E. is known to perform well for applications that are compute-intensive [150].

Option pricing is a basic financial model and is used extensively throughout the Financial Services sector for pricing European, American, Bermuda and various other options. An option is a contract between two parties (buyer/seller), where the buyer has the right but not the obligation to engage in a future transaction based on a financial instrument. The literature contains several publications related to parallel option pricing algorithms. Parallel algorithms for pricing various types of options using the multinomial lattice model are discussed in [32, 68, 59]. Optimizing this on the Cell requires communication and synchronization after every stage which leads to degraded performance. Option pricing using parallel algorithms based on Monte Carlo method are presented in [87, 112, 157].

Monte Carlo simulation is a popular technique used in financial markets to compute stock/asset prices, commodity prices and risk valuation that require estimating losses based on an underlying stochastic process. It also has wide application in computational physics, physical chemistry and computational biology. In this chapter, we design an efficient parallel pseudo-random number generator based on Mersenne Twister algorithm [94] and a quasi-random number generator based on the Hammersley Sequence [64]. For our implementation of Mersenne Twister, Cell achieves a speedup of over 11 as compared to the performance on current Intel and AMD architectures. We explore and analyze the performance of various normalization techniques such as Low Distortion Map (LDM) [125], Box Mueller transform [27] in Cartesian and Polar forms. Using these routines for Monte Carlo simulation we develop an efficient parallel implementation for pricing European options using the Black Scholes option pricing model.

Section 4.2.5 provides an extensive performance comparison of our implementation over NVIDIA

G80 using CUDA SDK and RapidMind Development Platform v2.1 for GPUs. The RapidMind development platform helps developers create high performance applications with less effort and low cost. We also compare the performance of our hand tuned code as compared to using RapidMind Development platform v2.1 for Cell. Our implementation achieves a speedup of 1.51 over NVIDIA G80 (using CUDA), a speedup of over 2 as compared with using RapidMind for Cell and a speedup of 1.26 as compared with using RapidMind for GPU. Our detailed analyses and performance results suggest that Cell is well suited for financial workloads.

4.2.1 Option pricing

Algorithm 4: Monte Carlo method for option pricing

Input: Current Price (S, Strike Price (K), Expiration time (T), Volatility (v), Yield rate (r), Number of cycles (N)

Output: $\hat{C}$: Discounted payoff value

1 **for** $j \leftarrow 1$ ***to*** N **do**

2 Generate uniform random number x;

3 Transform x to Gaussian (normal) random number $\hat{x}$;

4 Compute $C_j = S * e^{(r-0.5v^2)*T+v*\sqrt{T}*\hat{x}}$;

5 Compute $C_j = \max(0, C_j - K)$ for Call, $C_j = \max(0, K - C_j)$ for Put;

6 Average current option value $\hat{C} = e^{-rT} * \frac{1}{N}\Sigma_{j=1}^{N} C_j$;

An option is a contract between two parties where one party has the right but not the obligation to engage in a future transaction on an underlying security. Option pricing involves the computation of the option payoff value that depends on the current price of the underlying asset, expiration time, volatility of asset, and risk-free rate. The Black Scholes formula [25] is a celebrated model used to price options that is based on the assumption that the price of the underlying security follows the geometric Brownian Motion described as a continuous time stochastic differential equation. In a situation where there is no arbitrage the price can be calculated as the discounted expected value under the risk-neutral measure. Using the Monte Carlo method [96], this value can be calculated by averaging over a large number of sample values.

The pseudo-code for option pricing using the Monte Carlo method is given in Algorithm 4. In the option pricing algorithm, Steps 1 & 2 are the most computationally intensive steps, i.e. generating standard Gaussian (normal) random numbers. These are random numbers that have the

following probability density function.

$$f(x) = \frac{1}{2\pi} e^{-\frac{1}{2}x^2}$$

mean = 0, variance = 1

Option pricing is fundamental workload to the Financial Services Sector and is widely used to trade stock, commodity, bond and index options. Here, we observe that one of the most computationally intensive steps in using Monte Carlo simulation for financial modeling is the generation of standard Gaussian (normal) random numbers. To compute these we first compute uniform-pseudo/quasi random numbers and then transform them to standard normal random numbers using standard normalization techniques.

4.2.2 Random Number Generation

pseudo-random

The Monte Carlo method requires a high quality random number generator. We use the Mersenne Twister algorithm [94] as the pseudo-random number generator in our design. Figure 49 gives an illustration of the algorithm for (*N=624, M=397*). It uses an input seed to initialize an array of size N. This array is traversed in a round robin manner during the subsequent iterations of the algorithm. During each iteration, element i is updated using element $i + 1$ and $i + M$. A series of shift and bitwise operations on the i^{th} element gives the output random number.

There are two ways to parallelize this for the Cell. One technique is to optimize the algorithm for a single SPE and use different seeds for various SPEs to generate multiple random streams. Using a dynamic seed for each SPE ensures that the combined stream has high quality of randomness [95]. Another technique is to generate a single stream of random numbers using the various SPEs. It is important to note that in this algorithm the computation from the latter part of the array requires the updated data from the first part which makes the algorithm data dependent. To obtain high performance on Cell, we use the first parallelization technique in our design. However, using different seeds on different SPEs is not enough since the generated random numbers from the various SPEs may be correlated, leading to degraded quality of Monte Carlo simulations. A solution to this is Dynamic Creator [95] that is based on the Mersenne Twister algorithm. This generates different algorithm parameters for the various SPEs which helps in generating multiple independent streams.

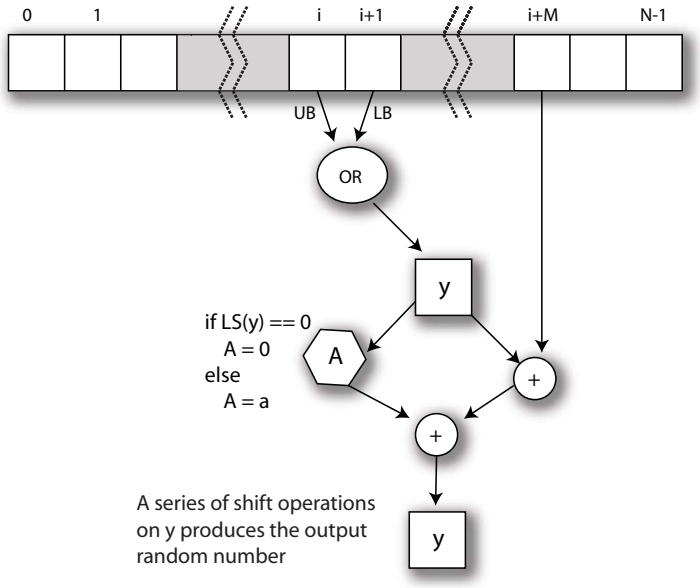

Figure 49: Illustration of the Mersenne Twister Algorithm. The function LS extracts the least significant bit from the input, and a is a constant in the algorithm.

Algorithm 5: Generating Hammersley point set

Input: Number of Simulations: N

Output: $\{(x_i, y_i)\}, i \in [1, N/2]$

//Divide loop iterations among p SPEs.

1 **for** $j \leftarrow 1$ ***to*** $\frac{N}{2}$ **do**

2 $\quad x_j = \frac{j}{N}$ //Unroll and vectorize for Cell optimization.

3 $\quad y_j = \frac{bit\_reverse(j)}{MAX\_INT}$

The data access pattern of the algorithm introduces challenges for optimizing this on the SPEs. For vectorization, the data access should be aligned to a 16 byte boundary. Calculating random number from array element i requires the value from element $i + M$, which may not be aligned in most cases. This requires using shuffle intrinsics within the SPE that degrades performance. In our implementation we use several techniques for optimization such as loop unrolling, branch hints, vectorization and use compare and select instructions to eliminate the branch in the algorithm.

quasi-random

Another technique uses the quasi Monte Carlo simulation for option pricing. This requires a quasi-random number generator.

In our approach we use the Hammersley sequence [64] that generates points uniformly within

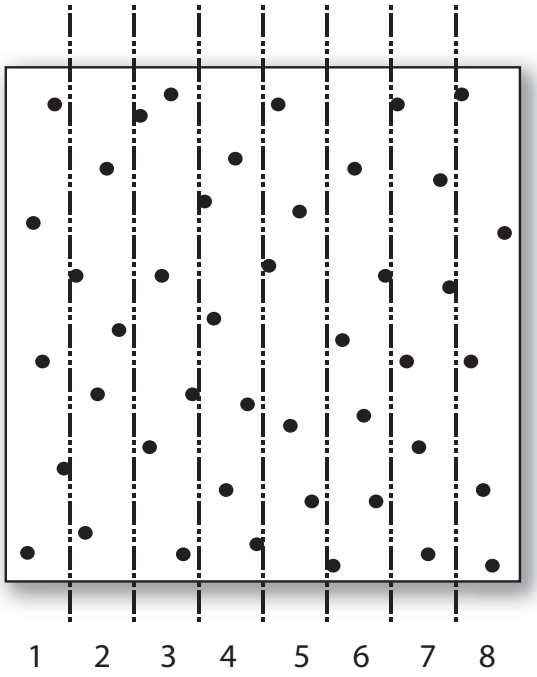

Figure 50: Illustration of the Cell parallelization for quasi-random number generation using Hammersley sequence. In this figure 8 SPEs are used for illustration. Here, SPE i is responsible for generating Hammersley points from the domain i.

Algorithm 6: Box Mueller transform in Cartesian form.

Input: Independent uniform random numbers (x,y)

Output: Normal random numbers ($\bar{x}$, $\bar{y}$)

1 $R = \sqrt{-2 * \ln x}$ //One multiplication, One logarithm, One square root

2 $\theta = 2\pi * y$ //One multiplication

3 $\bar{x} = R * \cos\theta$ //One multiplication, One trigonometric function

4 $\bar{y} = R * \sin\theta$ //One multiplication, One trigonometric function

Algorithm 7: Box Mueller transform in Polar form.

Input: Independent uniform random numbers (x,y)

Output: Normal random numbers ($\bar{x}$, $\bar{y}$)

1 $s = x^2 + y^2$ //Two multiplications, One addition

2 **if** $0 < s \leq 1$ **then**

3 $z = \sqrt{\frac{-2*\ln s}{s}}$ //One multiplication, One division, One square root, One logarithm

4 $\bar{x} = u * z$ //One multiplication

5 $\bar{y} = v * z$ //One multiplication

a square and has better statistical properties than many pseudo-random number generators. These sequences are known as low discrepancy sequences [106, 63].

The pseudo-code of the algorithm is given in Algorithm 5. This algorithm is computationally more expensive than Mersenne Twister, but is less branchy.

For parallelizing this on the Cell processor we divide the square domain into p equal parts (where p is the number of SPEs). SPE i generates Hammersley sequence from block i. Note that this algorithm generates a deterministic sequence and thus it is important to parallelize a single stream of random numbers. This is in contrast to the technique we used for parallelizing the Mersenne twister algorithm. Figure 50 gives an illustration of this technique. The x-coordinate of the domain is divided into p equal parts, where p is the number of SPEs. For each $x_j \in$ part i, SPE i generates the corresponding y_j.

4.2.3 Normalization

The random number generators discussed in the previous section generate uniform random numbers (random numbers uniformly distributed in the interval $[0, 1]$). The Monte Carlo approach for financial modeling (Algorithm 4) requires a random variable with Gaussian (normal) distribution (range $\in [-1, 1]$, mean = 0, variance = 1). In this section we analyze three techniques that could be used for transforming a set of uniform random numbers to normalized random numbers, and report their performance on the Cell processor.

Box Mueller transformation in Cartesian form

For every pair of input random numbers, Box Mueller transformation [27] in Cartesian form generates a pair of normalized random numbers. Algorithm 6 gives this transformation along with the computational effort required at each step.

For compute intensive operations such as log, $sqrt$, sin and cos we use the latest MASS (Mathematical Acceleration Subsystem) library that is available with Cell SDK 3.0. The MASS library routines take array inputs and give the best performance for large array sizes. We structure our implementation to provide long array inputs to the MASS routines in order to attain high performance.

Box Mueller transformation in Polar form

In the Polar form for every pair of input random numbers, a pair of normalized numbers is generated if the input pair lies within a unit disc. Algorithm 7 gives this transformation along with the computation effort required at each step.

In comparison to the Box Mueller transform in Cartesian form, this algorithm discards about one in four pairs of input random numbers, but it prevents the use of a trigonometric function (which is comparatively an expensive operation). Thus, Box Mueller in Polar form is a computationally less expensive as compared to the Cartesian form.

Algorithm 8: Extracting elements from array A that satisfy a condition X, using a vectorized approach

Input: array A, length N
Output: array C, number of extracted elements j

1 **for** $i \leftarrow 1$ ***to*** N **do**
2 $\quad B[i] = X(A[i])$ //Vectorize and Unroll for Cell optimization.
3 $j \leftarrow 0$
4 **for** $i \leftarrow 1$ ***to*** N **do**
5 $\quad C[j] = A[i]$ //Unroll for Cell optimization.
6 $\quad j = j + B[i]$

The presence of a branch in the algorithm poses issues during optimization on the Cell. The branch restricts vectorization of the algorithm. Also, due to the absence of a branch predictor on the SPEs it leads to a degradation in performance. This first problem reduces to extracting elements from a long input array A that satisfy a given condition X (let's call these 'good' elements), using vector intrinsics. To solve this problem in an elegant manner we create another array B, that stores the counter value for the corresponding element of A, i.e., If $A[i]$ is 'good', then $B[i]$ is 1 otherwise 0. Using B we extract the 'good' elements of A. The pseudo-code of this technique is given in Algorithm 8. Step 4 of the algorithm writes into the array C regardless of the value of $A[i]$. Note that this is correct as the array index of C increments only when a 'good' element is written to it. This leads to extra work but prevents branching from a conditional store. Although, Steps 4 & 5 of the algorithm are scalar, branch elimination significantly boosts the performance of the algorithm on the Cell processor. We use the MASS library for compute intensive operations such as $sqrt$ and log.

LDM : Low Distortion Map between square and disc

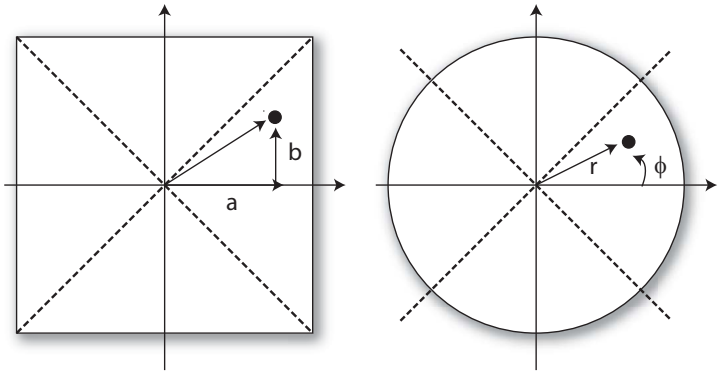

Figure 51: Illustration of the Low Distortion Map transformation. $r = a, \phi = \frac{\pi * b}{4 * a}$

Shirley and Chiu [125] describe a low distortion map (LDM) between a unit square and disk. This map preserves fractional area and introduces low distortion in shape. For a given point (a, b) within a unit square this transformation calculates values $r = a$, $\phi = \frac{\pi b}{4a}$. This maps to the output transformed point $(\bar{a}, \bar{b})$, where $a = r \cos \phi$, and $b = r \sin \phi$. Figure 51 gives an illustration of the algorithm. In our implementation we map every point in the unit square to the first quadrant and apply the LDM transformation. We use various optimization techniques such as vectorization, loop-unrolling and use the IBM MASS library to achieve high performance.

4.2.4 Optimizing option pricing for Cell

In Monte Carlo Simulation the number of cycles N (Algorithm 4) in general is very large, and the cycles are independent of one another. Thus, we divide the number of cycles among the various SPEs, with each SPE computing results from $\frac{N}{p}$ cycles, where p is the number of SPEs. We use our optimized kernels of random number generation and normalization as described earlier. Given the limited local store on an SPE pre-computing the normal random numbers and storing them on the PPE should be avoided. Instead, we calculate these numbers during each Monte Carlo cycle. Using Cell SIMD instructions we simultaneously calculate payoff values from four standard normal random numbers. Also for efficient pipeline utilization, we unroll the for loop in Algorithm 4 by a factor of 8.

The role of the PPE in the algorithm is to gather input data from the user, partition the work among the various SPEs (divide the total number of cycles), create SPE threads, gather the computed payoff value from each SPE, and average them to compute the final payoff value.

4.2.5 Performance Results

Table 10: Time in seconds to generate 100 million random samples in sequential and block pattern on various architectures. For the Cell/B.E. our timings are from a single chip. The performance results on the Intel, AMD and IBM PowerPC processors are from Saito and Matsumoto [120].

CPU/Compiler	Output	MT	MT(SIMD)
Intel Pentium-M 1.4 GHz	block	1.122	0.627
Intel C/C++ v9.0 [120]	seq	1.511	1.221
Intel Pentium-4 3.0 GHz	block	0.633	0.391
Intel C/C++ v9.0 [120]	seq	1.014	0.757
AMD Athlon 64 3800+	block	0.686	0.376
2.4 GHz, gcc v4.0.2 [120]	seq	0.756	0.607
IBM PowerPC G4 1.33 GHz	block	1.089	0.490
gcc v4.0.0 [120]	seq	1.794	1.358
IBM Cell/B.E. 3.2 GHz	block	-	0.034
xlc	seq	-	0.036

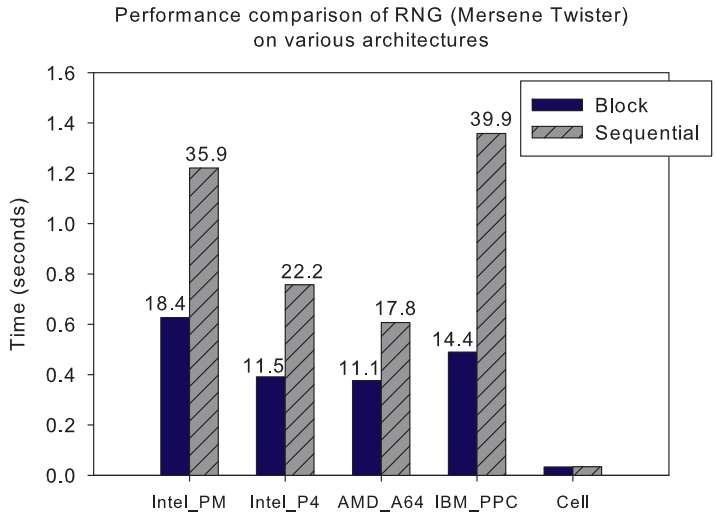

Figure 52: Comparison of running times to generate 100 million random samples in sequential and block pattern on various architectures as reported in Table 10. The number above each bar represents the speedup of Cell/B.E. as compared with the corresponding architecture.

We report our performance results from actual runs on a IBM BladeCenter QS20, with two 3.2 GHz Cell/B.E. processors, 512 KB Level 2 cache per processor, and 1 GB memory (512 MB per

processor). For performance comparisons we compile our code using the xlc compiler provided with Cell SDK 2.1, with level 3 optimization.

Table 10 lists the running time of our Mersenne Twister implementation on Cell and compares with other architectures. For performance comparisons with Intel, AMD and IBM PowerPC processors we use results from optimized implementations (using SIMD instructions) of the Mersenne Twister algorithm as reported by Saito and Matsumoto [120]. Figure 52 plots the performance and reports speedup of our Cell optimized implementation (using one Cell/B.E. processor) as compared to the corresponding architecture. Block approach generates a block of random numbers and Sequential approach generates one random number per iteration. *MT(SIMD)* gives the performance of a vectorized implementation of the Mersenne Twister algorithm. We achieve speedup of 11.5 over Intel Pentium 4, 3.0 GHz in the block random number generation and a speedup of 22.2 using the sequential approach.

Table 11: Performance comparison of option pricing using Monte Carlo simulation with other architectures. Mersenne Twister is used as the pseudo-random number generator.

Version	Platform	Performance	Transformation	Software
EOP-BMP (our result)	Cell/B.E.	**1040M/s**	Box Mueller /Polar form	Cell SDK 2.1
IBM SDK Sample	Cell/B.E.	190M/s	Box Mueller /Polar form	Cell SDK 2.1
RMCell-BMP	Cell/B.E.	605M/s	Box Mueller /Polar form	RapidMind SDK 2.1
EOP-BMC (our result)	Cell/B.E.	**1824M/s**	Box Mueller /Cartesian form	Cell SDK 2.1
CUDA-BMC [1]	NVIDIA G80 (GPU)	1209M/s	Box Mueller /Cartesian form	CUDA SDK 1.0

We use different combinations of random number generators and normalization techniques to compare performance across several platforms. For the remainder of this section we use both of the Cell/B.E. processors available on the blade for measuring performance. Table 11 gives a performance comparison of option pricing using Monte Carlo simulation with these architectures. The performance column reports the number of Monte Carlo experiments that can be performed per second. *EOP-BMP* demonstrates the performance of our implementation that uses Box Mueller in Polar form and *EOP-BMC* uses the Box Mueller in Cartesian form.

Table 12: Performance comparison of option pricing using quasi-Monte Carlo simulation with other architectures. Hammersley sequence is used as the quasi-random number generator.

Version	Platform	Performance	Transformation	Software
EOP-LDM (our work)	Cell/B.E.	**1770M/s**	Low Distortion Map (LDM)	Cell SDK 2.1
RMCell-LDM	Cell/B.E.	888M/s	Low Distortion Map (LDM)	RapidMind SDK 2.1
RMGPU-LDM	NVIDIA G80 (GPU)	1400M/s	Low Distortion Map (LDM)	RapidMind SDK 2.1

For *CUDA-BMC* we use an optimized implementation by Podlozhnyuk [116], that is based on the CUDA Software Development Toolkit, to show performance comparisons with NVIDIA G80. The code generates an array (domain set) of random samples, normalizes the array and uses that for pricing many options. For performance comparisons we aggregate the running time of all stages for pricing a single option. Our Cell optimized implementation (*EOP-BMC*) achieves a speedup of 1.51 over *CUDA-BMC*. The performance number for *RMCell-BMP* is based on implementation from IBM that uses the RapidMind development platform for optimizing option pricing on Cell. We change the normalization technique to optimize this code for performance comparisons. We observe that our hand-tuned code obtains a performance advantage of 1.72 as compared with using the RapidMind SDK for Cell. *IBM SDK Sample* gives the performance of an implementation of this algorithm provided as a sample with Cell SDK 2.1.

Table 12 gives a performance comparison of option pricing using quasi-Monte Carlo simulation with other architectures. *EOP-LDM* shows the performance of our Cell-optimized implementation based on Hammersley quasi-random number generator and the Low Distortion Map (LDM) transformation. *RMCell-LDM* represents the performance of the latest implementation from RapidMind Inc. of this algorithm compiled using RapidMind v2.1. *RMGPU-LDM* shows the performance of the same implementation run on NVIDIA GeForce 8800. RapidMind's performance on Cell is within a factor of 2 as compared to our hand-tuned implementation, and we obtain a speedup of 1.26 as compared to *RMGPU-LDM*.

4.3 Summary

The increasing rate of market data traffic is posing a very serious challenge to the financial industry and to the current capacity of trading systems worldwide. This requires solutions that protect the inherent nature of the business model, i.e., providing low processing latency with ever increasing capacity requirements. This chapter presents a novel solution to OPRA FAST feeds decoding and normalization, a piece of the larger mosaic, on commodity multicore processors. Our approach captures the essence of OPRA protocol specification in a handful of lines of DSParser, the high-level descriptive language that is the programming interface of the DotStar protocol parser, thus promising a solution that is high-performance, yet flexible and adaptive. We demonstrate impressive processing rates of 15 million messages per second on the fastest single socket Intel Xeon, and over 24 million messages per second using the IBM Power6 on a server with 4 sockets. We present an extensive performance evaluation that helps understand the intricacies of the decoding algorithm, and expose many distinct features of the emerging multicore processors, that can be used to estimate performance on these platforms.

We also design efficient parallel algorithms for European Option pricing on the Cell processor using Monte Carlo simulation. To achieve high performance, we design, analyze and optimize different high performance pseudo (such as Mersenne Twister) and quasi (such as Hammersley sequence) random number generators as well as normalization techniques, while maintaining high accuracy. Our Cell-optimized EO pricing attains a speedup of 1.51 over NVIDIA GeForce 8800 (using CUDA), a speedup of over 2 as compared to using RapidMind SDK for Cell and a speedup of 1.26 as compared to using RapidMind SDK for GPU. Our detailed analyses and performance results suggest that the IBM Cell/B.E. is well suited for financial workloads, and Monte Carlo simulation provides high scalability among the SPEs.

One evident limitation of this work is that it addresses only a part of the feed handler, leaving many important questions unanswered. For example, the network and the network stack are still areas of primary concern. Nevertheless, we believe that the initial results presented in this chapter are ground-breaking because they show that is possible to match or exceed speeds that are typical of specialized devices such as FPGAs, using commodity components and a high-level formalism

that allows quick configurability. We believe that the methodology presented in this chapter can be readily extended to the other parts of the ticker plant, in particular the value-added functionalities, and other data protocols taking full advantage of the multi- and many-core architectures that are expected to become a ubiquitous form of computing within the next few years.

CHAPTER V

PARALLEL DISCRETE DATA TRANSFORMATION ALGORITHMS

Preliminary versions of this chapter was published in:

- D.A. Bader, V. Agarwal, and S. Kang, "Computing Discrete Transforms on the Cell Broadband Engine," Parallel Computing, 35(3):119-137, 2009.
- D.A. Bader, V. Agarwal, "FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine," The 14th Annual IEEE International Conference on High Performance Computing (HiPC 2007), S. Aluru et al., (eds.), Springer-Verlag LNCS 4873, 172-184, Goa, India, December, 2007.

Data analysis is sometimes more intuitive when transformed to another format. Fourier transform is an operation that transforms one complex valued function of a real variable into another. Fast Fourier Transform (FFT) is an efficient algorithm that is used for computing the Discrete Fourier Transform. FFT is of primary importance and a fundamental kernel in many computationally intensive scientific applications such as computer tomography, data filtering and fluid dynamics. Another important application area of FFTs is in spectral analysis of speech, sonar, radar, seismic and vibration detection. FFTs are also used in digital filtering, signal decomposition, and in solution of partial differential equations. The performance of these applications rely heavily on the availability of a fast routine for Fourier transforms. In this chapter we present the design of an efficient parallel implementation of Fast Fourier Transform on the Cell Broadband Engine. More specifically this chapter presents the following contributions:

- *An efficient parallel algorithm, FFTC, for computing FFTs on the IBM Cell B.E., that uses an iterative out-of-place approach to solve 1D FFTs with 1K to 16K complex input samples*: We describe our methodology to partition the work among the SPEs to efficiently parallelize a *single FFT computation* where the source and output of the FFT are both stored in main memory.

- *Efficient synchronization barrier:* The algorithm requires a synchronization among the SPEs after each stage of FFT computation. Our synchronization barrier is designed to use inter SPE communication without any intervention from the PPE. The synchronization barrier requires only $2 \log p$ stages (p: number of SPEs) of inter SPE communication by using a tree-based approach. This significantly improves the performance, as PPE intervention not only results in a high communication latency but also in sequentialization of the synchronization step.

- *Scalable efficient parallel implementation of FFTC*: We achieve a performance improvement of over 4 as we vary the number of SPEs from 1 to 8. We attain a performance of 18.6 GFLOP/s for a single-precision FFT with 8K complex input samples and also show significant speedup in comparison with other architectures.

The rest of this chapter describes Fast Fourier transforms, prior work in this area, our FFTC algorithm and an extensive performance evaluation of FFTC on the IBM Cell. B.E.

5.1 Fast Fourier Transform

Fast Fourier Transform (FFT) is an efficient algorithm that is used for computing the Discrete Fourier Transform. Some of the important application areas of FFTs have been mentioned in the previous section. There are several algorithmic variants of the FFTs that have been well studied for parallel processors and vector architectures [2, 10, 11, 23].

In our design we utilize the naive Cooley-Tukey radix-2 Decimate in Frequency (DIF) algorithm. The pseudo-code for an out-of-place approach of this algorithm is given in Algorithm 9. The algorithm runs in $\log N$ stages and each stage requires O(N) computation, where N is the input size.

The array w contains the $twiddle\ factors$ required for FFT computation. At each stage the computed complex samples are stored at their respective locations thus saving a bit-reversal stage for output data. This is an iterative algorithm which runs until the parameter $problemSize$ reduces to 1. Figure 53 shows the butterfly stages of this algorithm for an input of 16 sample points (4 stages).

Apart from the theoretical complexity, another common performance metric used for the FFT algorithm is the floating point operation (FLOP) count. On analyzing the sequential algorithm,

we see that during each iteration of the innermost *for* loop there is one complex addition for the computation of first output sample, which accounts for 2 FLOPs. The second output sample requires one complex subtraction and multiplication which accounts for 8 FLOPs. Thus, for the computation of two output samples during each innermost iteration we require 10 FLOPs, which suggests that we require 5 FLOPs for the computation of a complex sample at each stage. The total computations in all stages are $N \log N$ which makes the total FLOP count for the algorithm as $5N \log N$.

Algorithm 9: Sequential FFT algorithm

Input: array $A[0]$ of size N

```
1  NP ⟵ 1 ;
2  problemSize ⟵ N;
3  dist ⟵ 1;
4  i1 ⟵ 0;
5  i2 ⟵ 1;
6  while problemSize > 1 do
7      Begin Stage;
8      a ⟵ A[i1];
9      b ⟵ A[i2];
10     k = 0, jtwiddle = 0;
11     for j ← 0 to N − 1 step 2 ∗ NP do
12         W ⟵ w[jtwiddle];
13         for jfirst ← 0 to NP do
14             b[j + jfirst] ← a[k + jfirst] + a[k + jfirst + N/2];
15             b[j + jfirst + Dist] ← (a[k + jfirst] − a[k + jfirst + N/2]) ∗ W;
16         k ← k + NP;
17         jtwiddle ← jtwiddle + NP;
18     swap(i1, i2);
19     NP ← NP ∗ 2;
20     problemSize ← problemSize/2;
21     dist ← dist ∗ 2;
22     End Stage;
```

Output: array $A[i1]$ of size N

5.1.1 Related Work

The formal description of Fourier theory and the Fast Fourier Transform can be found in a variety of articles and textbooks [72, 48, 107]. Over the past few decades many people have studied the Discrete Fourier Transform (DFT), and have presented algorithms for the fast computation of the

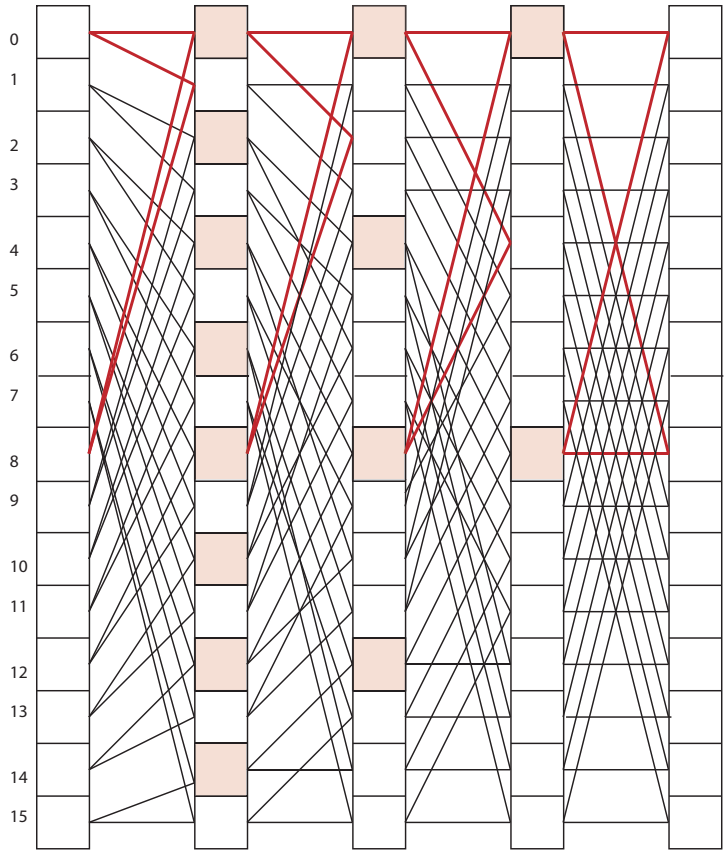

Figure 53: Butterflies of the ordered DIF FFT algorithm

DFT, such as Cooley-Tukey FFT algorithm [49], Prime-factor FFT algorithm [36], Brunn's FFT algorithm [30], Rader's FFT algorithm [117]. Several articles describe the vectorization of the FFT algorithm [2, 137, 1, 85]. There are other articles that present variations to the FFT algorithm [131, 31, 83, 113, 156]. Hardware-based FFTs have also been widely studied [89].

The literature also contains several publications related to FFTs on the Cell/B.E. processor and other multicore architectures. Table 13 provides an admittedly cursory overview of recent algorithms/implementations in this literature. We categorize these solutions with respect to the underlying algorithm used, size of FFT, performance as reported in the publication, architecture on which the algorithm has been tested for, and limitations.

5.2 FFTC: Our FFT Algorithm for the Cell B.E. Processor

There are several architectural features that make it difficult to optimize and parallelize the Cooley-Tukey FFT algorithm on the Cell Broadband Engine. The algorithm is branchy due to presence of a

Table 13: Related Work in Fast Fourier Transforms

Reference	Underlying algorithm	Size	Performance	Base architecture	Advantages/Limitations
Williams et al.[149]	naive radix-2 FFT algorithm	$4K$, $16K$, $64K$ complex 1D, $1K^2$, $2K^2$ complex 2D	30-42 GFlop/s (1D), 36-41 Gbps (2D), single precision	IBM Cell/B.E.	Performance Estimates.
Cico, Cooper and Greene [43]	single precision FFT algorithm	$4K$, $16K$, $64K$ complex 1D, $1K^2$, $2K^2$ complex 2D	22.1 GFlop/s for a single FFT, 176.8 GFlop/s for 8 FFTs in parallel ($8K$), 90.8 GFlop/s for $64K$	IBM Cell/B.E.	Code tuned for specific sizes, for $1K$ and $8K$ data resides in SPE for performance measurements.
Chow, Fossum and Brokenshire [42]	Cell-optimized vectorized radix-2 FFT algorithm	16M complex	46.8 GFLOP/s	IBM Cell/B.E.	Works only for input with 16M complex samples.
FFTW [57]	variety of FFT algorithms are used and are machine optimized	1D, 2D and 3D FFTs of various sizes	size specific	Intel/AMD/IBM/Sun commodity/multicore processors as well as IBM Cell/B.E.	User friendly library, but higher performance can be obtained than FFTW by tuning for specific FFT sizes.
Chellappa, Franchetti and Püeschel [39]	Spiral optimized Cooley-Tukey FFT algorithm	For DFT on multiple SPEs, 16-32K complex	5-36 GFlop/s single precision (for the input sizes in previous column)	IBM Cell/B.E.	Automatic optimized code generation using Spiral, large FFTs currently not supported.
Chen and Hu and Lin and Gao [40]	radix-2 Cooley-Tukey FFT algorithm	64K complex 1D, 256x256 complex 2D	20.72 GFlop/s (1D), 20 GFlop/s (2D)	IBM Cyclops-64	Performance estimates, specific to the given sizes.
Govindaraju and Manocha [62]	Stockham autosort Cooley-Tukey based FFT algorithm	0-8M complex values (1D)	16 GFlop/s for 4M single precision 1D	NVIDIA 8800 GPU	Limited by GPU/CPU interconnect

doubly nested *for* loop within the outer *while* loop. This results in a compromise on the performance due to the absence of a branch predictor on the Cell. The algorithm requires an array that consists of the $N/2$ complex twiddle factors. Since each SPE has a limited local store of 256 KB, this array cannot be stored entirely on the SPEs for a large input size. The limit in the size of the local store memory also restricts the maximum input data that can be transferred to the SPEs. Parallelization of a single FFT computation involves synchronization between the SPEs after every stage of the algorithm, as the input data of a stage is the output data of the previous stage. To achieve high performance it is necessary to divide the work equally among the SPEs so that no SPE waits at the synchronization barrier. Also, the algorithm requires $\log N$ synchronization stages which impacts the performance. It is difficult to vectorize every stage of the FFT computation. For vectorization of

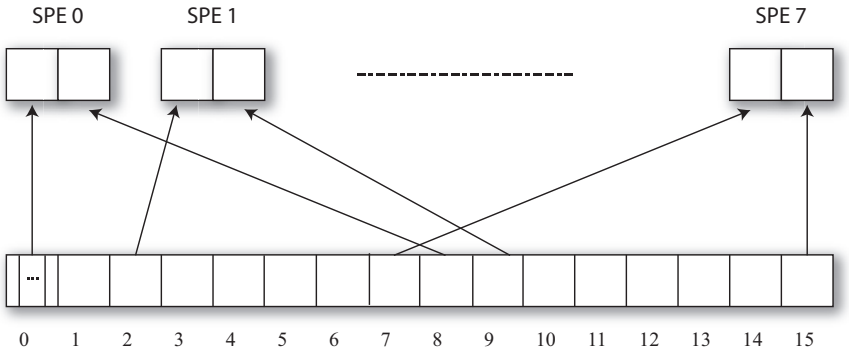

Figure 54: Partition of the input array among the SPEs (e.g. 8 SPEs in this illustration).

the first two stages of the FFT computation it is necessary to shuffle the output data vector, which is not an efficient operation in the SPE instruction set architecture. Also, the computationally intensive loops in the algorithm need to be unrolled for best pipeline utilization. This becomes a challenge given a limited local store on the SPEs.

5.2.1 Parallelizing FFTC for the Cell

As mentioned in the previous section for best performance it is important to partition work among the SPEs to achieve load balancing. We parallelize by dividing the input array held in main memory into $2p$ chunks, each of size $\frac{N}{2p}$, where p is the number of SPEs.

During every stage, SPE i is allocated chunk i and $i + p$ from the input array. The basis for choosing these chunks for an SPE lies in the fact that these chunks are placed at an offset of $N/2$

input elements. For the computation of an output complex sample we need to perform complex arithmetic operation between input elements that are separated by this offset. Figure 54 gives an illustration of this approach for work partitioning among 8 SPEs.

The PPE does not intervene in the FFT computation after this initial work allocation. After spawning the SPE threads it waits for the SPEs to finish execution.

5.2.2 Optimizing FFTC for the SPEs

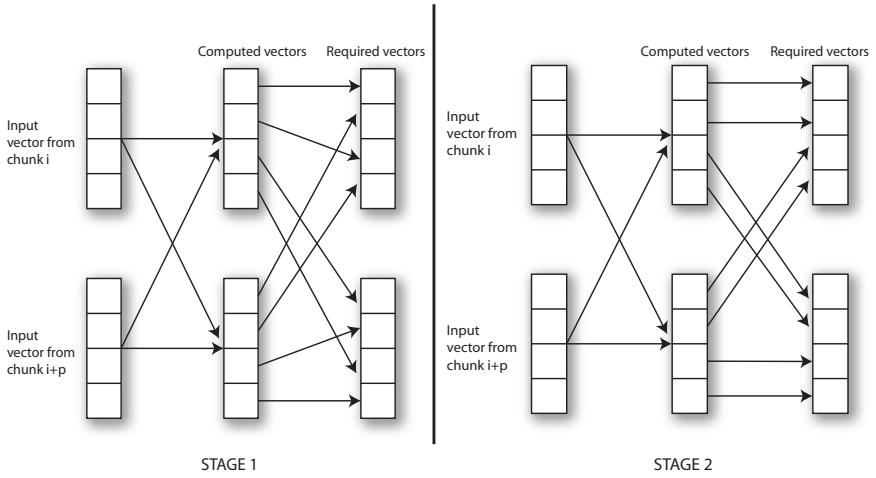

Figure 55: Vectorization of the first two stages of the FFT algorithm. These stages require a shuffle operation over the output vector to generate the desired output.

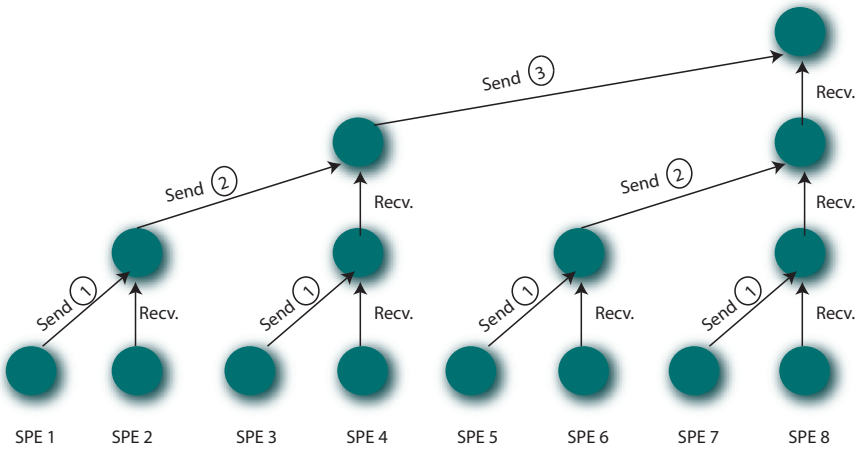

Figure 56: Stages of the synchronization barrier using inter SPE communication. The synchronization involves sending inter SPE mailbox messages up to the root of the tree and then sending back acknowledgment messages down to the leaves in the same topology.

After dividing the input array among the SPEs, each SPE is allocated 2 chunks each of size $\frac{N}{2p}$.

Each SPE, fetches this chunk from main memory using DMA transfers and uses double-buffering to overlap memory transfers with computation. Within each SPE, after computation of each buffer, the computed buffer is written back into main memory at offset using DMA transfers.

The detailed pseudo-code is given in Algorithm 10. The first two stages of the FFT algorithm are duplicated, that correspond to the first two iterations of the outer *while* loop in sequential algorithm. This is necessary as the vectorization of these stages requires a shuffle operation (*spu_shuffle()*) over the output to re-arrange the output elements to their correct locations. Please refer to Figure 55 for an illustration of this technique for stages 1 and 2 of the FFT computation.

The innermost *for* loop (in the sequential algorithm) can be easily vectorized for $NP > 4$, that correspond to the stages 3 through $\log N$. However, it is important to duplicate the outer *while* loop to handle stages where $NP <$ *buffersize*, and otherwise. The global parameter *buffersize* is the size of a single DMA get buffer. This duplication is required as we need to stall for a DMA transfer to complete, at different places within the loop for these two cases. We also unroll the loops to achieve better pipeline utilization. This significantly increases the size of the code thus limiting the unrolling factor.

SPEs are synchronized after each stage, using *inter-SPE communication*. This is achieved by constructing a binary synchronization tree, so that synchronization is achieved in $2 \log p$ stages. The synchronization involves the use of inter-SPE mailbox communication without any intervention from the PPE. Please refer to Figure 56 for an illustration of the technique.

This technique performs significantly better than other synchronization techniques that either use chain-like inter-SPE communication or require the PPE to synchronize between the SPEs. The chain-like technique requires $2p$ stages of inter-SPE communication whereas with the intervention of the PPE latency of communication reduces the performance of this barrier.

5.3 Performance Analysis of FFTC

For compiling, instruction level profiling, and performance analysis we use the IBM Cell Broadband Engine SDK 3.0.0-1.0, gcc 4.1.1 with level 3 optimization. From *'/proc/cpuinfo'* we determine the clock frequency as 3.2 GHz with revision 5.0. We use *gettimeofday()* on the PPE before computation on the SPE starts, and after it finishes. For profiling measurements we iterate the computation 10000

Algorithm 10: Parallel FFTC algorithm: View within SPE

Input: array in PPE of size N

Output: array in PPE of size N

```
1  NP ⟵ 1 ;
2  problemSize ⟵ N;
3  dist ⟵ 1;
4  fetchAddr ⟵ PPE input array;
5  putAddr ⟵ PPE output array;
6  chunkSize ⟵ N/(2*p);
7  Stage 0 (SIMDization achieved with shuffling of output vector);
8  Stage 1 ;
9  while NP < buffersize && problemSize > 1 do
10     Begin Stage;
11     Initiate all DMA transfers to get data;
12     Initialize variables;
13     for j ← 0 to 2 * chunkSize do
14         Stall for DMA buffer;
15         for i ← 0 to buffersize/NP do
16             for jfirst ← 0 to NP do
17                 SIMDize computation as NP > 4;
18             Update j, k, jtwiddle;
19         Initiate DMA put for the computed results
20     swap(fetchAddr, putAddr);
21     NP ← NP * 2;
22     problemSize ← problemSize/2;
23     dist ← dist * 2;
24     End Stage;
25     Synchronize using Inter-SPE communication;
26 while problemSize > 1 do
27     Begin Stage;
28     Initiate all DMA transfers to get data;
29     Initialize variables;
30     for k ← 0 to chunkSize do
31         for jfirst ← 0 to min(NP, chunkSize − k) step buffersize do
32             Stall for DMA buffer;
33             for i ← 0 to buffersize do
34                 SIMDize computation as buffersize > 4;
35             Initiate DMA put for the computed results;
36         Update j, k, jtwiddle;
37     swap(fetchAddr, putAddr);
38     NP ← NP * 2;
39     problemSize ← problemSize/2;
40     dist ← dist * 2;
41     End Stage;
42     Synchronize using Inter SPE communication;
```

times to eliminate the noise of the timer.

For parallelizing a single 1D FFT on the Cell, it is important to divide the work among the SPEs. Figure 57 shows the performance of our algorithm with varying the number of SPEs for 1K and 4K complex input samples. Our algorithm obtains a scalability of about 4x with 8 SPEs.

Our design requires a barrier synchronization among the SPEs after each stage of the FFT computation. We focus on FFTs that have from 1K to 16K complex input samples. For relatively small inputs and as the number of SPEs increases, the synchronization cost becomes a significant issue since the time per stage decreases but the cost per synchronization increases. With instruction level profiling we determine that the time required per synchronization stage using our tree-synchronization barrier is about 1 microsecond (3200 clock cycles). We achieve a high performance barrier using inter-SPE mailbox communication which significantly reduces the time to send a message, and by using the tree-based technique we reduced the number of communication stages required for the barrier ($2 \log p$ steps).

Figure 58 shows the single precision performance for complex inputs of FFTC, our optimized FFT, as compared with the following architectures:

- **IBM Power 5:** IBM OpenPower 720, Two dual-core 1.65 GHz POWER5 processors.
- **AMD Opteron:** 2.2 GHz Dual Core AMD Opteron Processor 275.
- **Intel Duo Core:** 3.0 GHz Intel Xeon Core Duo (Woodcrest), 4MB L2 cache.
- **Intel Pentium 4:** Four-processor 3.06 GHz Intel Pentium 4, 512 KB L2.

We use the performance numbers from benchFFT [57] for the comparison with the above architectures. We consider the FFT implementation that gives best performance on these architectures for comparison.

The Cell/B.E. has a two instruction pipelines, and for achieving high performance it is important to optimize the code so that the processor can issue two instructions per clock cycle. This level of optimization requires inspecting the assembly dump of the SPE code. For achieving pipeline utilization it is required that the gap between dependent instructions needs to be increased. We use the *IBM Assembly Visualizer for Cell/B.E.* tool to analyze this optimization. The tool highlights

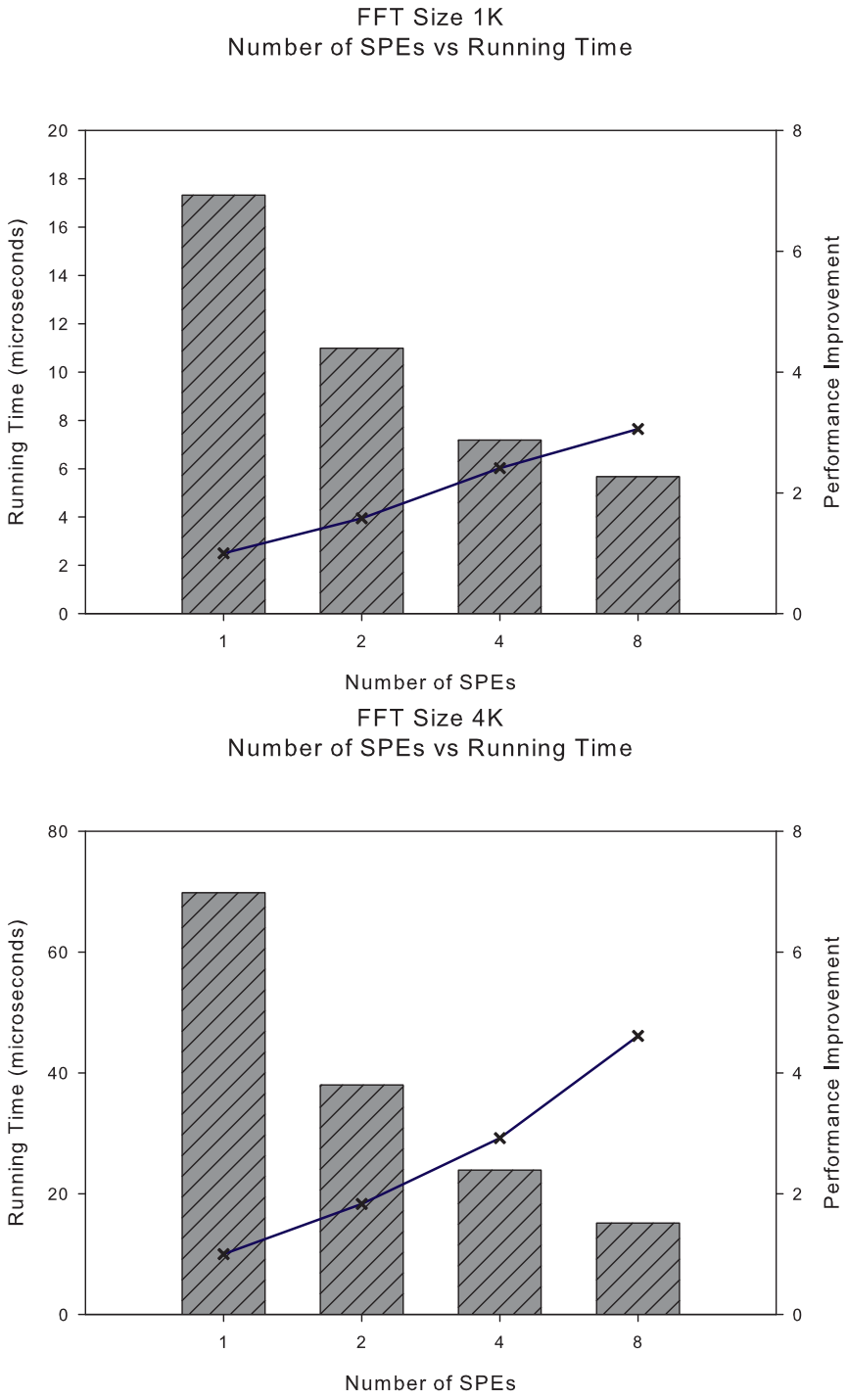

Figure 57: Running Time of our FFTC code on 1K and 4K inputs as we increase the number of SPEs.

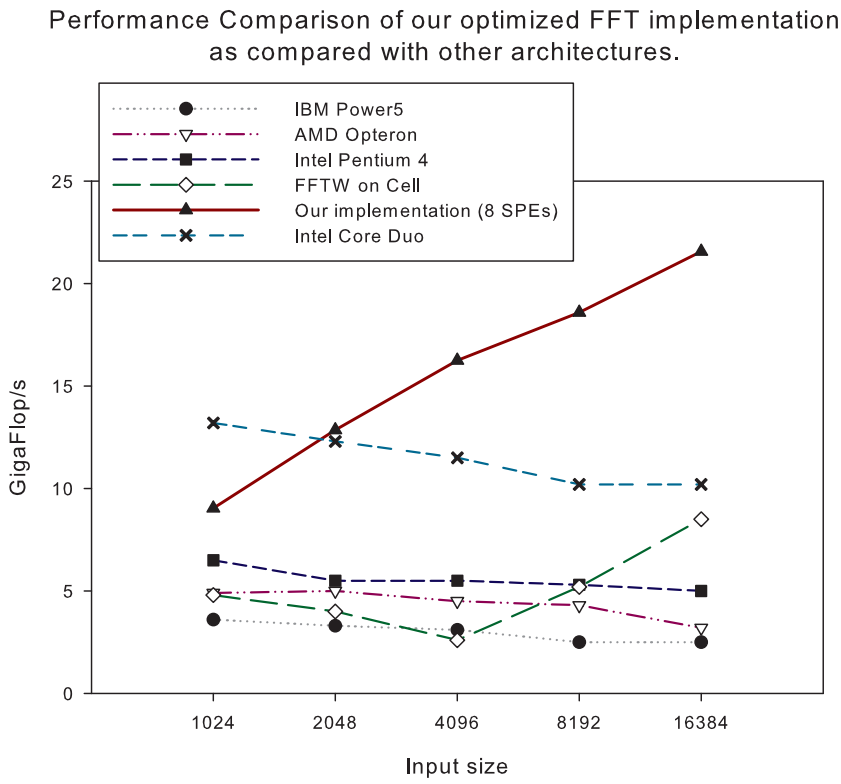

Figure 58: Performance comparison of FFTC with other architectures for various input sizes of FFT. The performance numbers are from benchFFT from the FFTW website.

the stalls in the instruction pipelines and helps the user to reorganize the code execution while maintaining correctness. Figure 59 shows the analysis of pipeline utilization. The left and right column contains the instruction distribution among the two pipelines. For each instruction the 'x' in the column shows which cycle the execution starts for that instruction and the 'x's below show the number of cycles the instruction takes to execute in the pipeline. Some portions utilize these pipelines effectively (top figure) whereas there are a few stalls in other parts of the code which still need to be optimized (bottom figure).

5.4 Summary

In summary, in this chapter we design an efficient parallel transformation algorithm for data analysis. These class of algorithms have deterministic data access and computation patterns that can be exploited to achieve optimized high level algorithmic design on multicore processors. We present

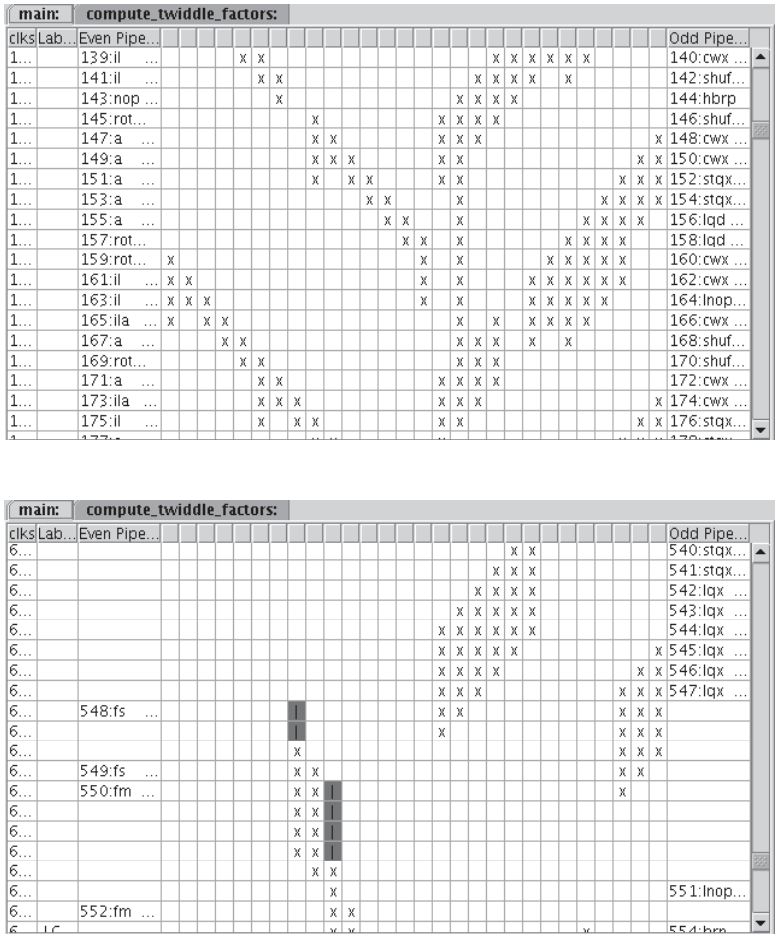

Figure 59: Analysis of the pipeline utilization using the *IBM Assembly Visualizer for Cell Broadband Engine*. The top figure shows full pipeline utilization for certain parts of the code and the bottom figure shows areas where the pipeline stalls due to data dependency.

FFTC, our high-performance design to parallelize the 1D FFT on the Cell Broadband Engine processor. FFTC uses an iterative out-of-place approach and we focus on FFTs with 1K to 16K complex input samples. We describe our methodology to partition the work among the SPEs to efficiently parallelize a single FFT computation. The computation on the SPEs is fully vectorized with other optimization techniques such as loop unrolling and double buffering. The algorithm requires a synchronization among the SPEs after each stage of FFT computation. Our synchronization barrier is designed to use inter SPE communication only without any intervention from the PPE. The synchronization barrier requires only $2 \log p$ stages (p: number of SPEs) of inter SPE communication by using a tree-based approach. This significantly improves the performance, as PPE intervention not

only results in a high communication latency but also results in sequentializing the synchronization step. We achieve a performance improvement of over 4 as we vary the number of SPEs from 1 to 8. We also demonstrate FFTC's performance of 18.6 GFLOP/s for an FFT with 8K complex input samples and show significant speedup in comparison with other architectures. Our implementation outperforms Intel Duo Core (Woodcrest) for input sizes greater than 2K and to our knowledge we have the fastest FFT for these range of complex input samples.

CHAPTER VI

CONCLUSIONS

We present efficient parallel algorithms, that can process massive-data sets and streams, from several application areas in network analysis, security and computational finance, and design scalable high performance parallel implementations that are optimized for the emerging multicore processors. Our algorithmic design captures the machine-independent aspects, to guarantee portability with performance to future processors, and our implementations embeds processor-specific optimizations. Current multicore processors have a number of processing cores integrated on to a single chip. The processing cores have their own private L1 cache (or a hierarchy of private caches), a shared common L2 cache and shared main memory. Caches are small, and memory access from the cache is signicantly faster than access from the main memory. Some processors also contain hardware capabilities for in-core parallelism based on vector units. In future, these processors are expected to have the same high-level design and may appear as building blocks of exascale systems, thus designing parallel algorithms that exploit the hardware capabilities of these processors and optimize inter-processor communication becomes critical for sustaining high performance computing.

Designing efficient parallel algorithms on multicore processors requires orchestration between the complex compute and communication requirements of the algorithm with the capabilities and features that the hardware has to offer. Compute units work in stages, the execution delays varies from instruction to instruction. Thus getting best performance involves re-structuring the computation of the algorithm to use efficient instructions, and providing the compute-units with enough independent instructions to fill all stages of the pipeline. The vector units add another level of parallelism in-core, but require deterministic sequence of compute instructions on all elements of the vector. Branches in algorithms create bottlenecks in finding such sequences, as compute decisions are made at runtime. To optimize and reduce branches in the algorithm, we observe that it is often cheaper to do something than to decide to do nothing. When computation involved within the branch is very less, the speculative execution of those instructions is sometimes cheaper than the

average branch misprediction penalty.

When the algorithm uses a small data structure the memory access pattern has little effect on overall performance. However, for large data structures regular memory access patterns help exploit the cache on the processor and exhibit good performance, but irregular access patterns cause significant degradation in performance due to large memory access latencies. When access can be predicted in advance, data structures can either be re-designed to aid cache locality or data can be prefetched dynamically at runtime. Also, with increasing data rates and memory requirements of the algorithm, the limited cache size and memory necessitates designing more compact data structures. Multicore systems have large shared memory but access to different parts of the memory comes at different costs, and the cache coherency protocol also causes significant delays when data is accessed across sockets. Parallel algorithms that need frequent communication between cores at runtime need efficient communication paradigms, and algorithms should be re-designed to restrict access to local memory banks as much as possible. An efficient algorithm is one that is designed by looking at the problem at a high-level, finds the right tradeoff between these various processor parameters and captures machine independent aspects to ensure scalability and portability to future systems.

In Chapter 2 we discussed data analysis using graph traversal. Graphs are often used to represent data in security, biology and network analysis. Graph-theoretic algorithms generally have very little computation, and highly irregular memory access patterns. A processor with large fast shared memory would perform very well for such algorithms, but since modern multicore procesors are limited by a small fast cache, access to the slow main memory should be limited as much as possible. For graph algorithms predicting the data access pattern is generally not feasible. Data structures are large and parallelism in these algorithms is fine-grained. Breadth-first search (BFS) and List ranking are representative kernels of many memory intensive combinatorial applications. We have presented a first scalable breadth-first search (BFS) algorithm for commodity multicore processors. In spite of the highly irregular access pattern of the BFS, our algorithm was able enforce various degrees of memory and processor locality, minimizing the negative effects of the cache-coherency protocol between processor sockets. We design an innovative data layout that enhances memory locality and cache utilization through a well-defined hierarchy of working sets; a smaller frequently accessed

data structure that exploits cache locality, and a larger infrequently accessed graph structure for edge traversal. We also partition the graph and other data structures across sockets, and allow only local access to them. We design an efficient, low-latency channel mechanism for inter-socket communication that tolerates the potentially high delays of the cache-coherency protocol. The experimental results, conducted on two Intel Nehalem platforms, a dual-socket Nehalem EP and a four-socket Nehalem EX, have demonstrated an impressive processing rate in parsing graphs that have up to a billion edges. Using several graph configurations the Nehalem EX system has reached, and in many cases exceeded, the performance of special-purpose supercomputers designed to handle irregular applications. These are significant results in parallel computing as prior results in graph traversal report very limited or no speedup on irregular graphs when compared to their best sequential implementation. Madduri [91] presented parallel graph algorithms that use high performance supercomputing systems.

We also develop a fast parallel implementation of the list ranking algorithm for the Cell processor using a generic work partitioning technique and present a first result that shows the usefulness of this architecture for graph-theoretic algorithms. Our work partitioning technique is build over software-managed threads that help hide memory latency of irregular memory accesses. We confirm the efficacy of our technique by demonstrating an improvement factor of 4.1 as we tune the number of software managed threads per core. Most importantly we demonstrate an overall speedup of 8.34 of our implementation over an efficient PPE-only sequential implementation. We show substantial speedups by comparing the performance of our list ranking implementation with several single processor and parallel architectures.

Analyzing streaming data in-core is impractical with modern processors due to the increasing data rates over the network. With rates more than 8 Gbits per second current general purpose technology provides only a handful of clock cycles to process every incoming byte. Thus designing efficient algorithms that process streaming data in parallel becomes critical in this domain. There is no doubt that the growing network traffic is exposing serious bottlenecks of the current network intrusion detection systems. This problem is exaggerated by the large number of signatures that need to be scanned concurrently and at line rate. A practical solution to the keyword scanning problem not only must be as fast as possible, but also must address many other dimensions: it

should be able to parse strings of arbitrary length, should be storage-efficient, should not degrade performance when we increase the number of patterns, should be resilient to attacks, should not waste computational resources, should be portable and exhibit parallel scalability, should allow fast dynamic updates and, above all, should map directly to the native mechanisms of the available hardware to allow for an efficient implementation. Keyword scanning algorithms construct a large state automata based on an NFA or DFA. DFAs are space in-efficient and require a full array of state transitions, NFA are runtime in-efficient, where each input character may require several state transitions on average to determine next state. Therefore, NFAs and DFAs in their original form are not scalable solutions in both space and speed. Also, the runtime algorithm over state automatas is a graph traversal algorithm that comes with design challenges as mentioned previously. We present first such comprehensive solution and design an innovative algorithmic solution for keyword scanning on network data that is able to achieve space efficiency and high speed with very large input dictionaries, even when the system is under attack. We design (a) a compression algorithm that divides the states of the automaton into two parts: a cache of frequently accessed states and the remaining states that are expressed as a "linear combination" of the cached states, (b) a model of computation based on small CAMs (Content Addressable Memory) that can be efficiently mapped on processor architectures that provide vector extensions, and (c) a data layout that enforces data re-use and minimizes memory traffic, with a careful orchestration of the memory requests; the data layout aggregates segments of memory that are likely to be accessed in sequence within the same unit of transfer, such as a cache line. Our framework is composed by a pattern compiler that takes a dictionary and optional training input to generate an automaton, other optimized data structures, and an efficient run-time system that takes this automaton to parse input data streams. Our optimized implementations on a range of high-end Intel x86 processors, achieve a processing rate of 16 Gbit/sec in a dual-socket configuration under heavy hitting. We have also presented an extensive evaluation that explores the scalability of our algorithm to millions of keywords, the performance obtained with various input and dictionary characteristics, and demonstrated the resilience of the framework under network attacks.

The increasing rate in streaming market data traffic is posing a very serious challenge to the

financial industry and to the current capacity of trading systems worldwide. This requires solutions that protect the inherent nature of the business model, i.e., providing low processing latency with ever increasing capacity requirements. Financial institutions receive market data feeds in compressed and encoded format. Processing these feeds requires identifying fields of several categories, decoding them individually and processing the information contained within. The structure of the message is specified by the market data exchange. Thus, intuitively this problem reduces to identifying known knowledge structures from incoming data using a fast pattern recognition system, and then calling optimized field processing kernels. Due to the smaller set of knowlege structures the pattern recognition problem is much simpler than what we encounter in network security, but obtaining low latency under a portable framework is difficult. We present a novel solution to OPRA FAST feeds decoding and normalization, on commodity multicore processors. Our approach captures the essence of OPRA protocol specification in a handful of lines of DSParser, the high-level descriptive language that is the programming interface of the DotStar protocol parser, thus promising a solution that is high-performance, yet flexible and adaptive. We demonstrate impressive processing rates of 15 million messages per second on the fastest single socket Intel Xeon, and over 24 million messages per second using the IBM Power6 on a server with 4 sockets. These processing rates are more than an order of magnitude faster than the current needs of the market. We present an extensive performance evaluation that helps understand the intricacies of the decoding algorithm, and expose many distinct features of the emerging multicore processors, that can be used to estimate performance on these platforms. The increasing data rates and number of financial instruments also puts pressure on pricing engines and analytics that are used by financial institutions to manage risk and their market positions. We present an optimized implementation on the Cell processor of European option pricing based on Monte Carlo simulation, that uses optimized kernels for generating pseudo and quasi random numbers.

Data transformation techniques are often employed for more intuitive understanding of the data sets. We also present a high-performance design to parallelize the 1D Fast Fourier transform (FFT) on the Cell B.E. processor. Our algorithm uses an iterative out-of-place approach and we focus on FFTs with 1K to 16K complex input samples. We describe our methodology to partition the work among the Cell cores to efficiently parallelize a single FFT computation. The computation on

the SPEs is fully vectorized with other optimization techniques such as loop unrolling and double buffering. As the FFT computation requires a synchronization step after every stage of the algorithm, we design an efficient synchronization barrier that uses only inter SPE communication. We demonstrate a performance of 18.6 GFLOP/s for an FFT with 8K complex input samples and show significant speedup in comparison with other architectures. Our implementation outperforms Intel Duo Core (Woodcrest) for input sizes greater than 2K and most other implementations present in literature for this range of complex input samples.

We are working to extend our Breadth-first search algorithm, to achieve load balancing for graphs with skewed degree distributions. We have designed techniques that modify existing data structure and use that information to create work buckets. We encode the size of a node within the next address of each edge. This reduces an extra memory access per node that reads node size in the load balancing routine which is an expensive operation in multicore processors. Using our technique we expect to extend our results and analyze more complex graphs. We are also working to analyze the requirements for scaling our current BFS implementation to a system comprising of a large number of multicore processors. This analysis will also help analyze the design requirements of future exascale systems. One such idea involves designing a novel communication network that is built using the concept of our communication channels to provide seamless low latency communication between the sockets of the large HPC system. Designing efficient algorithms for pattern recognition that use large number of complex mining (hundreds of thousands to millions) patterns based on regular expressions is another area that requires ground breaking research contributions. The automatas built using earlier algorithms presented in literature will either create very large data sets or result in state explosion that current multicore systems will be incapable to handle in memory. The techniques we presented in this dissertation to design parallel algorithms for keyword scanning maybe a good starting point for this problem. Our design of a novel feed handler is a small piece of a large ticker plant that financial institutions require to conduct high frequency and algorithmic trading. Other parts of the ticker plant as described in earlier chapters also add significant latency to the entire pipeline and need efficient solutions either based on commodity or special purpose technology. Also, analyzing patterns from market news feeds to aid intelligent trading would require extensive research in designing knowledge structures and high performance algorithms to process

the feeds at real time. Algorithm design challenges will be aggravated by larger exascale systems that will contain millions of processing cores. We believe that the results presented in this dissertation forms a valuable foundation to develop the architectural and algorithmic building blocks of upcoming exascale machines. Innovative solutions to process and analyze data will keep emerging with the advancement of science, technology and better understanding of the universe, we believe that this research area will keep presenting the scientific community with very challenging problems in the future.

REFERENCES

[1] AGARWAL, R. C. and COOLEY, J. W., "Fourier transform and convolution subroutines for the IBM 3090 Vector facility," *IBM J. Res. Dev.*, vol. 30, no. 2, pp. 145–162, 1986.

[2] AGARWAL, R. C. and COOLEY, J. W., "Vectorized mixed radix discrete Fourier transform algorithms," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1283–1292, 1987.

[3] AGARWAL, V., BADER, D. A., DAN, L., LIU, L.-K., PASETTO, D., PERRONE, M., and PETRINI, F., *Attaining High Performance Communication: A Vertical Approach*, ch. The Case of the Fast Financial Feed. Chapman & Hall/CRC, September 2009.

[4] AGARWAL, V., BADER, D. A., DAN, L., LIU, L.-K., PASETTO, D., PERRONE, M., and PETRINI, F., "Faster FAST: Multicore Acceleration of Streaming Financial Data," *Computer Science - R&D*, vol. 23, no. 3-4, pp. 249–257, 2009.

[5] AGARWAL, V., LIU, L.-K., and BADER, D. A., "Financial Modeling on the Cell Broadband Engine," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS '08)*, (Miami, FL), pp. 1–12a, March 2008.

[6] AGARWAL, V., PETRINI, F., PASETTO, D., and BADER, D. A., "Scalable Graph Exploration on Multicore Processors," in *Proc. Supercomputing (SC '10)*, (New Orleans, LA), November 2010.

[7] AHO, A. V. and CORASICK, M. J., "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[8] AMD, "Multicore processors from AMD," 2010.

[9] ASANOVIC, K. and *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[10] ASHWORTH, M. and LYNE, A. G., "A segmented FFT algorithm for vector computers," *Parallel Computing*, vol. 6, no. 2, pp. 217–224, 1988.

[11] AVERBUCH, A., GABBER, E., GORDISSKY, B., and MEDAN, Y., "A parallel FFT on an MIMD machine," *Parallel Computing*, vol. 15, pp. 61–74, 1990.

[12] BADER, D. A. and CONG, G., "A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors (SMPs)," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS '04)*, (Santa Fe, NM), April 2004.

[13] BADER, D. A. and CONG, G., "Fast Shared-Memory Algorithms for Computing the Minimum Spanning Forest of Sparse Graphs," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS '04)*, (Santa Fe, NM), April 2004.

[14] BADER, D. A., CONG, G., and FEO, J., "A Comparison of the Performance of List Ranking and Connected Components Algorithms on SMP and MTA Shared-Memory Systems," tech. rep., Electrical and Computer Engineering Department, The University of New Mexico, Albuquerque, NM, October 2004.

[15] BADER, D. A., ILLENDULA, A. K., MORET, B. M. E., and WEISSE-BERNSTEIN, N., "Using PRAM algorithms on a uniform-memory-access shared-memory architecture," in *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE '01)* (BRODAL, G. S., FRIGIONI, D., and MARCHETTI-SPACCAMELA, A., eds.), vol. 2141, (Århus, Denmark), pp. 129–144, 2001.

[16] BADER, D. A. and MADDURI, K., "GTgraph: A Synthetic Graph Generator Suite," 2006.

[17] BADER, D. A., SRESHTA, S., and WEISSE-BERNSTEIN, N., "Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs)," in *Proc. 9th Int'l Conf. on High Performance Computing (HiPC '02)* (SAHNI, S., PRASANNA, V., and SHUKLA, U., eds.), vol. 2552, (Bangalore, India), pp. 63–75, December 2002.

[18] BADER, D. A. and AGARWAL, V., "FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine," in *Proc. Int'l Conf. on High Performance Computing (HiPC '07)*, (Goa, India), pp. 172–184, December 2007.

[19] BADER, D. A., AGARWAL, V., and KANG, S., "Computing discrete transforms on the Cell Broadband Engine," *Parallel Computing*, vol. 35, no. 3, pp. 119–137, 2009.

[20] BADER, D. A., AGARWAL, V., and MADDURI, K., "On the Design and Analysis of Irregular Algorithms on the Cell Processor: A case study on list ranking," in *21st Intl. Parallel and Distributed Processing Symp. (IPDPS '07)*, (Long Beach, CA), March 2007.

[21] BADER, D. A., AGARWAL, V., MADDURI, K., and KANG, S., "High performance combinatorial algorithm design on the Cell Broadband Engine processor," *Parallel Computing*, vol. 33, no. 10-11, pp. 720–740, 2007.

[22] BADER, D. A. and MADDURI, K., "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2," in *Proc. 35th Int'l Conf. on Parallel Processing (ICPP'06)*, (Columbus, OH), pp. 523–530, IEEE Computer Society, August 2006.

[23] BAILEY, D. H., "A high-performance FFT algorithm for vector supercomputers," *Int'l Journal of Supercomputer Applications*, vol. 2, no. 1, pp. 82–87, 1988.

[24] BARROSO, L. A., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B., SMITH, S., STETS, R., and VERGHESE, B., "Piranha: a scalable architecture based on single-chip multiprocessing," in *Proc. of the 27th Annual Int'l Symp. on Computer Architecture (ISCA '00)*, (Vancouver, British Columbia, Canada), pp. 282–293, June 2000.

[25] BLACK, F. and SCHOLES, M., "Monte Carlo methods for solving multivariate problems," *The Journal of Political Economy*, vol. 81, no. 3, pp. 637–654, 1973.

[26] BLAGOJEVIC, F., STAMATAKIS, A., ANTONOPOULOS, C., and NIKOLOPOULOS, D., "RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Engine," in *Int'l Parallel and Distributed Processing Symp. (IPDPS '07)*, (Long Beach, CA), March 2007.

[27] BOX, G. E. P. and MULLER, M. E., "A note on the generation of random normal deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 1958.

[28] BOYER, R. S. and MOORE, J. S., "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[29] BROKENSHIRE, D. A., "Maximizing the power of the Cell Broadband Engine Processor: 25 tips to optimal application performance," tech. rep., IBM, June 2006.

[30] BRUUN, G., "z-transform DFT filters and FFT's," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 26, pp. 56–63, February 1978.

[31] BURRUS, C. and ESCHENBACHER, P., "An in-place, in-order prime factor FFT algorithm," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 29, pp. 806–817, August 1981.

[32] CAMPOLIETI, G. and MAKAROV, R., "Parallel lattice implementation for option pricing under mixed state-dependent volatility models," in *Proc. 19th Int'l Symp. on High Performance Computing Systems and Applications (HPCS '05)*, (Guelph, Ontario, Canada), pp. 170–176, May 2005.

[33] CARLSON, W. W., DRAPER, J. M., CULLER, D. E., BROOKS, K. Y. E., and WARREN, K., "Introduction to UPC and Language Specification," Tech. Rep. CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[34] CARON, P. and ZIADI, D., "Characterization of Glushkov Automata," *Theoretical Compututer Science*, vol. 233, no. 1-2, pp. 75–90, 2000.

[35] CHAKRABARTI, D., ZHAN, Y., and FALOUTSOS, C., "R-MAT: A recursive model for graph mining," in *Proc. 4th SIAM Int'l Conf. on Data Mining (SDM '04)*, (Lake Buena Vista, FL), April 2004.

[36] CHAN, S. C. and HO, K. L., "On indexing the prime factor fast Fourier transform algorithm," *IEEE Transactions on Circuits and Systems*, vol. 38, pp. 951–953, August 1991.

[37] CHANDRAMOWLISHWARAN, A., WILLIAMS, S., OLIKER, L., LASHUK, I., BIROS, G., and VUDUC, R., "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures," in *Proc. IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS'10)*, (Atlanta, GA), April 2010.

[38] CHANDRAMOWLISHWARAN, A., KNOBE, K., and VUDUC, R., "Performance evaluation of Concurrent Collections on high-performance multicore computing systems," tech. rep., Georgia Institute of Technology, February 2010.

[39] CHELLAPPA, S., FRANCHETTI, F., and PÜESCHEL, M., "Computer generation of fast fourier transforms for the cell broadband engine," in *Proc. 23rd Int'l Conf. on Supercomputing (ICS '09)*, pp. 26–35, June 2009.

[40] CHEN, L., HU, Z., LIN, J., and GAO, G. R., "Optimizing the Fast Fourier Transform on a Multi-core Architecture," in *Int'l Parallel and Distributed Processing Symp. (IPDPS '07)*, (Long Beach, CA), pp. 1–8, March 2007.

[41] CHEN, T., RAGHAVAN, R., DALE, J., and IWATA, E., "Cell Broadband Engine Architecture and its first implementation," tech. rep., IBM, November 2005.

[42] CHOW, A. C., FOSSUM, G. C., and BROKENSHIRE, D. A., "A Programming Example: Large FFT on the Cell Broadband Engine," *Proc. Global Signal Processing Expo (GSPx)*, 2005.

[43] CICO, L., COOPER, R., and GREENE, J., "Performance and Programmability of the IBM/-Sony/Toshiba Cell Broadband Engine Processor." White paper, 2006.

[44] CLAUSET, A., NEWMAN, M. E. J., and MOORE, C., "Finding Community Structure in Very Large Networks," *Physical Review E*, vol. 6, p. 066111, December 2004.

[45] COFFMAN, T., GREENBLATT, S., and MARCUS, S., "Graph-based technologies for intelligence analysis," *Commun. ACM*, vol. 47, no. 3, pp. 45–47, 2004.

[46] COLE, R. and VISHKIN, U., "Faster optimal prefix sums and list ranking," *Information and Computation*, vol. 81, no. 3, pp. 344–352, 1989.

[47] COMMENTZ-WALTER, B., "A String Matching Algorithm Fast on the Average," in *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, (London, UK), pp. 118–132, Springer-Verlag, 1979.

[48] COOLEY, J., LEWIS, P., and WELCH, P., "Application of the fast Fourier transform to computation of Fourier integrals, Fourier series, and convolution integrals," *IEEE Transactions on Audio and Electroacoustics*, vol. 15, pp. 79–84, June 1967.

[49] COOLEY, J. W. and TUKEY, J. W., "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[50] COPPERSMITH, D. and WINOGRAD, S., "Matrix multiplication via arithmetic progressions," in *Proc. 19th Annual ACM Symposium on Theory of Computing*, pp. 1–6, ACM, 1987.

[51] CRAY INC., "The XMT platform," 2006.

[52] DELORIMIER, M., KAPRE, N., MEHTA, N., RIZZO, D., ESLICK, I., RUBIN, R., URIBE, T. E., KNIGHT, T. F. J., and DEHON, A., "GraphStep: A System Architecture for Sparse-Graph Algorithms," in *Proc. Symp. on Field-Programmable Custom Computing Machines (FCCM '06)*, (Los Alamitos, CA, USA), IEEE Computer Society, 2006.

[53] DHARMAPURIKAR, S., KRISHNAMURTHY, P., SPROULL, T. S., and LOCKWOOD, J. W., "Deep Packet Inspection using Parallel Bloom Filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004.

[54] DUCH, J. and ARENAS, A., "Community Detection in Complex Networks Using extremal Optimization," *Physical Review E*, vol. 72, January 2005.

[55] ERDOGAN, O. and CAO, P., "Hash-AV: fast virus signature scanning by cache resident filters," *Int. J. Secur. Netw.*, vol. 2, no. 1/2, pp. 50–59, 2007.

[56] FLACHS, B., ASANO, S., DHONG, S. H., HOFSTEE, P., GERVAIS, G., KIM, R., LE, T., LIU, P., LEENSTRA, J., LIBERTY, J., MICHAEL, B., OH, H., MUELLER, S. M., TAKAHASHI, O., HATAKEYAMA, A., WATANABE, Y., and YANO, N., "A streaming processor unit for a Cell processor," in *Int'l Solid State Circuits Conference*, vol. 1, (San Fransisco, CA, USA), pp. 134–135, February 2005.

[57] FRIGO, M. and JOHNSON, S. G., "FFTW on the Cell Processor," tech. rep., 2007.

[58] GAZIT, H. and MILLER, G. L., "An improved parallel algorithm that computes the BFS numbering of a directed graph," *Inf. Process. Lett.*, vol. 28, no. 2, pp. 61–65, 1988.

[59] GERBESSIOTIS, A. V., "Architecture independent parallel binomial tree option price valuations," *Parallel Computing*, vol. 30, no. 2, pp. 301–316, 2004.

[60] GIACOMONI, J., MOSELEY, T., and VACHHARAJANI, M., "FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue," in *Proc. 13th Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, (Salt Lake City, UT), February 2008.

[61] GLUSHKOV, V. M., "The abstract theory of automata," *Russian Mathematical Survey. v16. 1-53*, 1961.

[62] GOVINDARAJU, N. K. and MANOCHA, D., "Cache-efficient numerical algorithms using graphics hardware," *Parallel Computing*, vol. 33, no. 10-11, pp. 663–684, 2007.

[63] HALTON, J. H., "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals," *Num. Math.*, vol. 2, no. 1, pp. 84–90, 1960.

[64] HAMMERSLEY, J., "Monte Carlo methods for solving multivariable problems," in *Proc. of the New York Academy of Science*, vol. 86, pp. 844–874, 1960.

[65] HELMAN, D. R. and JÁJÁ, J., "Designing practical efficient algorithms for symmetric multiprocessors," in *Algorithm Engineering and Experimentation (ALENEX'99)*, vol. 1619, (Baltimore, MD), pp. 37–56, Jan. 1999.

[66] HELMAN, D. R. and JÁJÁ, J., "Prefix computations on symmetric multiprocessors," *Journal Parallel & Distributed Computing*, vol. 61, no. 2, pp. 265–278, 2001.

[67] HPCWIRE, "Wall Street-HPC Lovefest; Intel's Fall Classic." White paper, September 2007.

[68] HUANG, K. and THULASIRAM, R. K., "Parallel algorithm for pricing American Asian options with multi-dimensional assets," in *Proc. 19th Int'l Symp. on High Performance Computing Systems and Applications (HPCS '05)*, (Guelph, Ontario, Canada), pp. 177–185, May 2005.

[69] HUTCHINGS, B. L., FRANKLIN, R., and CARVER, D., "Assisting network intrusion detection with reconfigurable hardware," *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM '02)*, pp. 111–120, April 2002.

[70] INOKUCHI, A., WASHIO, T., and MOTODA, H., "An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data," in *Proc. 4th European Conf. on Principles of Data Mining and Knowledge Discovery (PKDD '00)*, (Lyon, France), pp. 13–23, September 2000.

[71] INTEL, "Intel Multicore Technology," 2010.

[72] JACKSON, L. B., *Signals, systems, and transforms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1991.

[73] JACOBI, C., OH, H. J., TRAN, K. D., COTTIER, S. R., MICHAEL, B. W., NISHIKAWA, H., TOTSUKA, Y., NAMATAME, T., and YANO, N., "The Vector Floating-Point Unit in a Synergistic Processor Element of a Cell Processor," in *Proc. 17th IEEE Symp. on Computer Arithmetic*, (Cape Cod, MA), pp. 59–67, June 2005.

[74] JÁJÁ, J., *An Introduction to Parallel Algorithms*. New York: Addison-Wesley Publishing Company, 1992.

[75] JOHNSON, T. and NAWATHE, U., "An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2)," in *Proc. Int'l Symp. on Physical Design (ISPD '07)*, (Austin, TX), pp. 2–2, 2007.

[76] KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., and SHIPPY, D., "Introduction to the Cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[77] KAMIL, S., CHAN, C., OLIKER, L., SHALF, J., and WILLIAMS, S., "An Auto-Tuning Framework for Parallel Multicore Stencil Computations," in *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS '10)*, (Atlanta, GA), April 2010.

[78] KARP, R. M. and RABIN, M. O., "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.

[79] KARYPIS, G., HAN, E.-H. S., and KUMAR, V., "Chameleon: Hierarchical clustering using dynamic modeling," *Computer*, vol. 32, no. 8, pp. 68–75, 1999.

[80] KISTLER, M., PERRONE, M., and PETRINI, F., "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006.

[81] KLEIN, P. N. and SUBRAMANIAN, S., "A randomized parallel algorithm for single-source shortest paths," *J. Algorithms*, vol. 25, no. 2, pp. 205–220, 1997.

[82] KLIMOVITSKI, A., "Using SSE and SSE2: Misconceptions and Reality," *Developer UPDATE Magazine, Intel*, vol. 1, pp. 1–8, March 2001.

[83] KOLBA, D. and PARKS, T., "A prime factor FFT algorithm using high-speed convolution," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 25, pp. 281–294, Aug 1977.

[84] KONGETIRA, P., AINGARAN, K., and OLUKOTUN, K., "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[85] KORNAND, D. G. and LAMBIOTTE, J. J., "Computing the Fast Fourier Transform on a Vector Computer," *Mathematics of Computation*, vol. 33, pp. 977–992, 1979.

[86] KREBS, V. E., "Mapping networks of terrorist cells," *Connections*, vol. 24, no. 3, pp. 43–52, 2001.

[87] LI, J. X. and MULLEN, G. L., "Parallel computing of a quasi-Monte Carlo algorithm for valuing derivatives," *Parallel Computing*, vol. 26, no. 5, pp. 641–653, 2000.

[88] LIN, H.-A. P., "Estimation of the optimal performance of ASN.1/BER transfer syntax," *SIGCOMM Comput. Commun. Rev.*, vol. 23, no. 3, pp. 45–58, 1993.

[89] LIN, Y. T., TSAI, P. Y., and CHIUEH, T. D., "Low-power variable-length fast Fourier transform processor," *IEEE Proc. Computers and Digital Techniques*, vol. 152, pp. 499–506, July 2005.

[90] LIU, Q., LIANG, Z., GE, Y., ZAO, H., and JOSEPH, D. J., "String Matching for Large Dictionaries on a Heterogenous Multicore Processor," tech. rep., IBM, 2008.

[91] MADDURI, K., *A High-Performance Framework for Analyzing Massive Complex Networks*. PhD thesis, Georgia Institute of Technology, 2008.

[92] MADDURI, K., WILLIAMS, S., ETHIER, S., OLIKER, L., SHALF, J., STROHMAIER, E., and YELICKY, K., "Memory-efficient optimization of Gyrokinetic particle-to-grid interpolation for multicore processors," in *Proc. of the Conf. on High Performance Computing Networking, Storage and Analysis (SC'09)*, (Portland, Oregon), pp. 1–12, November 2009.

[93] MARTENS, W., NEVEN, F., and SCHWENTICK, T., "Complexity of Decision Problems for Simple Regular Expressions," in *Proc. Mathematical Foundations of Computer Science (MFCS '04)*, (Prague, Czech Republic), pp. 889–900, August 2004.

[94] MATSUMOTO, M. and NISHIMURA, T., "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[95] MATSUMOTO, M. and NISHIMURA, T., "Dynamic creation of pseudorandom number generators," in *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pp. 56–69, Springer, 2000.

[96] METROPOLIS, N. and ULAM, S., "The Monte Carlo Method," *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, 1949.

[97] MIZELL, D. and MASCHHOFF, K., "Early experiences with large-scale Cray XMT systems," in *Proc. 24th Int'l Symp. on Parallel & Distributed Processing (IPDPS'09)*, (Rome, Italy), May 2009.

[98] MOLKA, D., HACKENBERG, D., SCHONE, R., and MULLER, M. S., "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, (Raleigh, NC), pp. 261–270, September 2009.

[99] MORET, B. M. E., BADER, D. A., and WARNOW, T., "High-performance algorithm engineering for computational phylogenetics," *J. Supercomput.*, vol. 22, no. 1, pp. 99–111, 2002.

[100] MUTH, R. and MANBER, U., "Approximate Multiple Strings Search," in *Proc. 7th Annual Symp. on Combinatorial Pattern Matching (CPM '96)*, (Laguna Beach, CA), pp. 75–86, June 1996.

[101] MUTHUKRISHNAN, S., "Data streams: algorithms and applications," in *Proc. of the fourteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '03)*, (Philadelphia, PA, USA), pp. 413–413, Society for Industrial and Applied Mathematics, 2003.

[102] NAVARRO, G. and RAFFINOT, M., *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. New York, NY, USA: Cambridge University Press, 2002.

[103] NEWMAN, M. E. J., "Detecting Community Structure in Networks," *European Physical Journal B*, vol. 38, pp. 321–330, May 2004.

[104] NEWMAN, M. E. J., "Fast Algorithm for Detecting Community Structure in Networks," *Physical Review E*, vol. 69, p. 066133, June 2004.

[105] NEWMAN, M. E. J. and GIRVAN, M., "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, p. 026113, February 2004.

[106] NIEDERREITER, H., *Random number generation and quasi-Monte Carlo methods*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992.

[107] ORFANIDIS, S. J., *Introduction to signal processing*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[108] OVERELL, P., "RFC2234: Augmented BNF for Syntax Specifications (ABNF)," 1997.

[109] PANG, R., PAXSON, V., SOMMER, R., and PETERSON, L., "binpac: a yacc for writing application protocol parsers," in *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, (New York, NY, USA), pp. 289–300, ACM, 2006.

[110] PASETTO, D., PETRINI, F., and AGARWAL, V., "DotStar: breaking the scalability and performance barriers in parsing regular expressions," *Computer Science - R&D*, vol. 25, no. 1-2, pp. 93–104, 2010.

[111] PASETTO, D., PETRINI, F., and AGARWAL, V., "Tools for very fast regular expression matching," *IEEE Computer*, vol. 43, no. 3, pp. 50–58, 2010.

[112] PAULETTO, G., "Parallel Monte Carlo methods for derivative security pricing," in *Revised Papers from the Second Int'l Conf. on Numerical Analysis and Its Applications (NAA '00)*, (Rousse, Bulgaria), pp. 650–657, Springer-Verlag, June 2001.

[113] PEASE, M. C., "An Adaptation of the Fast Fourier Transform for Parallel Processing," *J. ACM*, vol. 15, no. 2, pp. 252–264, 1968.

[114] PETRINI, F., AGARWAL, V., and PASETTO, D., "SCAMPI: a scalable CAM-based algorithm for multiple pattern inspection," in *Proc. Supercomputing (SC '09)*, (Portland, OR), pp. 1–11, November 2009.

[115] PHAM, D., ASANO, S., BOLLIGER, M., DAY, M. N., HOFSTEE, H. P., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., RILEY, M., SHIPPY, D., STASIAK, D., SUZUOKI, M., WANG, M., WARNOCK, J., WEITZEL, S., WENDEL, D., YAMAZAKI, T., and YAZAWA, K., "The design and implementation of a first-generation Cell processor," in *International Solid State Circuits Conference*, vol. 1, (San Fransisco, CA, USA), pp. 184–185, February 2005.

[116] PODLOZHNYUK, V., "Monte Carlo Option pricing." (NVIDIA CUDA) White paper, v1.0, June, 2007.

[117] RADER, C. M., "Discrete Fourier transforms when the number of data samples is prime," *Proceedings of the IEEE*, vol. 56, pp. 1107–1108, June 1968.

[118] RAMASWAMY, R., KENCL, L., and IANNACCONE, G., "Approximate fingerprinting to accelerate pattern matching," in *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, (New York, NY, USA), pp. 301–306, ACM, 2006.

[119] REID-MILLER, M., "List Ranking and List Scan on the CRAY C-90," *J. Comput. Syst. Sci.*, vol. 53, pp. 344–356, Dec. 1996.

[120] SAITO, M. and MATSUMOTO, M., "Simple and Fast MT: A Two times faster new variant of Mersenne twister," in *Proc. 7th Intl. Conference on Monte Carlo Methods in Scientific Computing*, (Germany), 2006.

[121] SALMELA, L., TARHIO, J., and KYTÖJOKI, J., "Multipattern string matching with q-grams," *J. Exp. Algorithmics*, vol. 11, p. 1.1, 2006.

[122] SCARPAZZA, D. P., VILLA, O., and PETRINI, F., "Peak-Performance DFA-based String Matching on the Cell Processor," in *Proc. Int'l Workshop on System Management Techniques, Processes and Services (SMTPS '07)*, (Long Beach, CA), March 2007.

[123] SCARPAZZA, D. P., VILLA, O., and PETRINI, F., "Efficient Breadth-First Search on the Cell/BE Processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 10, pp. 1381–1395, 2008.

[124] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., and HANRAHAN, P., "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[125] SHIRLEY, P. and CHIU, K., "A low distortion map between disk and square," *Journal of graphics tools*, vol. 2, no. 3, pp. 45–52, 1997.

[126] SIAC, "FAST for OPRA - v1," 2007.

[127] SIAC, "Data Recipient Specification," 2008.

[128] SIAC, "FAST for OPRA - v2," 2008.

[129] SIAC, "National Market Systems - Common IP Multicast Distribution Network - Recipient Interface Specification," 2008.

[130] SIDHU, R. and PRASANNA, V. K., "Fast Regular Expression Matching Using FPGAs," *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pp. 227–238, 2001.

[131] SINGLETON, R., "An algorithm for computing the mixed radix fast Fourier transform," *Audio and Electroacoustics, IEEE Transactions on*, vol. 17, pp. 93–103, Jun 1969.

[132] SRIDHARAN, S., RODRIGUES, A., and KOGGE, P., "Evaluating Synchronization Techniques for Light-weight Multithreaded/Multicore Architectures," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, (New York, NY, USA), pp. 57–58, ACM, 2007.

[133] SRINIVASAN, R., "RPC: Remote Procedure Call Protocol Specification Version 2," 1995.

[134] SRINIVASAN, R., "XDR: External Data Representation Standard," 1995.

[135] SUBRAMANIAM, V. and CHENG, P. R. K.-H., "A Fast Graph Search Multiprocessor Algorithm," in *Proc. of the Aerospace and Electronics Conf. (NAECON'97)*, (Dayton, OH), July 1997.

[136] SUD, A., ANDERSEN, E., CURTIS, S., LIN, M. C., and MANOCHA, D., "Real-time Path Planning for Virtual Agents in Dynamic Environments," in *IEEE Virtual Reality*, (Charlotte, NC), March 2007.

[137] SWARZTRAUBER, P. N., "Vectorizing the FFTs," *Parallel Computations*, 1982.

[138] TABB GROUP, "Trading At Light Speed: Analyzing Low Latency Data Market Data Infrastructure," 2007.

[139] THORUP, M., "Undirected single-source shortest paths with positive integer weights in linear time," *J. ACM*, vol. 46, no. 3, pp. 362–394, 1999.

[140] TUCK, N., SHERWOOD, T., CALDER, B., and VARGHESE, G., "Deterministic memory-efficient string matching algorithms fo intrusion detection," in *Proceedings of the IEEE Infocom Conference*, pp. 333–340, 2004.

[141] ULLMAN, J. and YANNAKAKIS, M., "High-probability parallel transitive closure algorithms," in *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA '90)*, (Island of Crete, Greece), pp. 200–209, ACM, July 1990.

[142] VAN LUNTEREN, J., "High-Performance Pattern-Matching for Intrusion Detection," in *INFOCOM*, 2006.

[143] VASILIADIS, G., ANTONATOS, S., POLYCHRONAKIS, M., MARKATOS, E. P., and IOANNIDIS, S., "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, (Berlin, Heidelberg), pp. 116–134, Springer-Verlag, 2008.

[144] VAZQUEZ, A., FLAMMINI, A., MARITAN, A., and VESPIGNANI, A., "Global protein function prediction in protein-protein interaction networks," *Nature Biotechnology*, vol. 21, no. 6, pp. 697–700, 2003.

[145] VILLA, O., SCARPAZZA, D. P., PETRINI, F., and PEINADOR, J. F., "Challenges in Mapping Graph Exploration Algorithms on Advanced Multi-core Processors," in *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS '07)*, (Long Beach, CA), March 2007.

[146] VILLA, O., CHAVARRIA, D., and MASCHHOFF, K., "Input-independent, Scalable and Fast String Matching on the Cray XMT," *IEEE International Symposium on Parallel and Distributed Processing, 2009 (IPDPS 2009).*, May 2009.

[147] VILLA, O., SCARPAZZA, D., and PETRINI, F., "Accelerating Real-Time String Searching with Multicore Processors," *IEEE Computer*, vol. 41, pp. 42–50, April 2008.

[148] WEINSBERG, Y., TZUR-DAVID, S., DOLEV, D., and ANKER, T., "High performance string matching algorithm for a network intrusion prevention system (NIPS)," in *Workshop on High Performance Switching and Routing*, (Poznan, Poland), June 2006.

[149] WILLIAMS, S., SHALF, J., OLIKER, L., KAMIL, S., HUSBANDS, P., and YELICK, K., "The potential of the Cell processor for scientific computing," in *Proc. 3rd Conference on Computing Frontiers (CF '06)*, (New York, NY, USA), pp. 9–20, ACM Press, 2006.

[150] WILLIAMS, S., SHALF, J., OLIKER, L., KAMIL, S., HUSBANDS, P., and YELICK, K., "Scientific computing kernels on the Cell processor," *In'l Journal of Parallel Programming*, vol. 35, no. 3, pp. 263–298, 2007.

[151] WILLIAMS, S., WATERMAN, A., and PATTERSON, D., "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures," *Communications of the ACM (CACM)*, vol. 52, April 2009.

[152] WU, S. and MANBER, U., "A fast algorithm for multi-pattern searching," Tech. Rep. TR-94-17, Department of Computer Science, University of Arizona, 1994.

[153] XIA, Y. and PRASANNA, V. K., "Topologically Adaptive Parallel Breadth-first Search on Multicore Processors," in *Proc. 21st Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS'09)*, (Cambridge, MA), November 2009.

[154] YOO, A., CHOW, E., HENDERSON, K., MCLENDON, W., HENDRICKSON, B., and CATALYUREK, U., "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," in *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'05)*, (Seattle, WA), IEEE Computer Society, 2005.

[155] ZHANG, L., KIM, Y. J., and MANOCHA, D., "A Simple Path Non-Existence Algorithm using C-Obstacle Query," in *Proc. Intl. Workshop on the Algorithmic Foundations of Robotics (WAFR'06)*, (New York City), July 2006.

[156] ZHONG, C. X., HAN, G. Q., and HUANG, M. H., "Some new parallel fast fourier transform algorithms," in *Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on*, pp. 624–628, Dec. 2005.

[157] ZHU, W., THULASIRAMAN, P., THULASIRAM, R. K., and GAO, G. R., "Exploring financial applications on many-core-on-a-chip architecture: A first experiment.," in *ISPA Workshops*, vol. 4331 of *Lecture Notes in Computer Science*, pp. 221–230, Springer, 2006.